Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO FINAL DE GRADO

TÍTULO DEL TFG: Uso de tarjetas GPU para acelerar el procesado de señales

TITULACIÓN: Grado en Ingeniería de Sistemas de Telecomunicación

AUTOR: David Amat Sanz

DIRECTOR: Gabriel Montoro López

CODIRECTOR: Pere Lluis Gilabert Pinal

FECHA: 05 de Julio de 2017

Título: Uso de tarjetas GPU para acelerar el procesado de señales

Autor: David Amat Sanz

Director: Gabriel Montoro López

Codirector: Pere Lluis Gilabert Pinal

Fecha: 05 de Julio de 2017

Resumen

Este Trabajo de Final de Grado pretende analizar y poner en práctica otra forma de procesar señales digitales, mejorando su rendimiento y velocidad de ejecución.

Los DSP y FPGA son los elementos más usados en la actualidad para cualquier tipo de procesado de señales. Este proyecto se centra en el uso de las tarjetas gráficas (GPU) para explotar al máximo el paralelismo del que se dispone hoy en día.

Los procesadores actuales (CPU) cuentan con unos pocos núcleos y trabajan secuencialmente lo que puede comportar elevado consumo de tiempo si se está procesando cantidades muy grandes de datos.

La ventaja de las GPU es que disponen de miles de núcleos, que, pese a ser menos potentes individualmente, si trabajan de forma simultánea pueden ofrecer una notable aceleración respecto a la CPU.

El algoritmo Least Mean Square, usado en filtros adaptativos, es el escogido para llevar a cabo la paralelización.
Su complejidad computacional es ideal para esta optimización, pues, como hay que encontrar los coeficientes de adaptación mediante un gran número de iteraciones, vamos a poder realizar todas estas multiplicaciones a la vez con CUDA C, que es el lenguaje que se utiliza en computación paralela para tarjetas gráficas de la marca Nvidia.

Title: Use of GPU cards to speed up signal processing

Author: David Amat Sanz

Director: Gabriel Montoro López

Codirector: Pere Lluis Gilabert Pinal

Date: July 05 th 2017

Overview

This Bachelor's Degree Final Project aims to analyze and implement another way to process digital signals, improving their performance and speed of execution.

DSP and FPGA are the most commonly used elements for any kind of signal processing. This project focuses on the use of graphics cards (GPU) to exploit to the maximum the parallelism that is available today.

Current processors (CPUs) have a few cores and work sequentially which can be very time consuming if large amounts of data are being processed.

The advantage of GPUs is that they have thousands of cores, which, although they are less powerful individually, if they work simultaneously they can offer a remarkable acceleration compared to the CPU.

The Least Mean Square algorithm, used in adaptive filters, is the one chosen to perform the parallelization.
Its computational complexity is ideal for this optimization, because, as we have to find the adaptation coefficients through a large number of iterations, we will be able to perform all these multiplications at the same time with CUDA C, which is the language used in parallel computing for Nvidia graphics cards.

Quiero agradecer a toda mi familia, amigos y a mi pareja por apoyarme desde el inicio. Su motivación me ha ayudado a lograr los objetivos que perseguía. También agradecer tanto a mi tutor Gabriel Montoro como a mi cotutor Pere Lluis Gilabert por la ayuda ofrecida durante todo el desarrollo del proyecto.

ÍNDICE

INTRODUCCIÓN

En la actualidad los filtros adaptativos constituyen una importante parte del procesamiento digital de señales.

El uso de estos filtros ofrece una atractiva solución al problema que usualmente no pueden solventar los filtros de coeficientes fijos convencionales.

Los filtros de coeficientes constantes no suelen ser suficientes a la hora de tratar señales que son variantes en el tiempo o que son parte de entornos contaminados, debido a que trabajan en frecuencias fijas y por tal motivo se ven limitados en el procesamiento adecuado de la señal.

Combinar un filtro con un algoritmo adaptativo, genera posibilidades de que el sistema de filtrado se ajuste a cualquier señal de entrada, esto permite recalcular los coeficientes del filtro de una forma dinámica y efectiva, convirtiéndolo de esta manera en un filtro adaptativo.

En este trabajo se propone una versión paralela del algoritmo LMS (Least Mean Square Algorithm), que se utiliza para procesamiento de señales digitales como la eliminación de eco y reducción de ruido para la obtención de una señal deseada.

El paralelismo en la GPU (Unidad de procesamiento gráfico) permite la descomposición de un problema en una serie de problemas más pequeños y que pueden ser calculados más rápidamente.

Los resultados obtenidos, especialmente el aumento de la velocidad y la eficiencia, muestran que el método paralelo implementado en la GPU es mucho más rápido que otros procedimientos existentes que se llevan a cabo solamente en la CPU, pues esta solo trabaja con unos cuantos núcleos optimizados para el procesamiento en serie secuencial.

CAPÍTULO 1. Tipos de filtros y elección del adaptativo

1.1. Filtro electrónico

Un filtro electrónico se define como un dispositivo electrónico, que tiene por objetivo aislar, dejar pasar o eliminar un subconjunto de señales de un conjunto de las mismas, siendo este dispositivo, selectivo en frecuencia, específico y predecible.
El objetivo es extraer una determinada información de interés.

1.1.1. Filtros analógicos

Un filtro analógico es un sistema de tiempo continuo que obedece a una ecuación diferencial lineal con coeficientes constantes y que modifica las componentes frecuenciales de una señal analógica de forma diferente en función de su frecuencia.

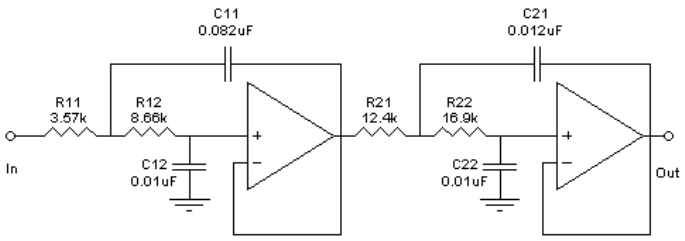


Fig. 1.1 Ejemplo de filtro Butterworth de orden 4 (Fc = 1KHz)

1.1.2. Filtros digitales

Un filtro digital es cualquier procedimiento que permite transformar los datos digitalizados en otros datos mediante un determinado algoritmo.

Los filtros digitales realizan la función del filtro a través de algoritmos numéricos. El proceso se realiza mediante el diagrama de bloques siguiente:

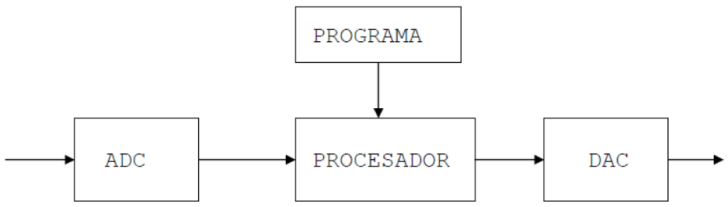


Fig. 1.2 Diagrama del proceso de un filtro digital

Se caracterizan por ser sistemas predecibles, flexibles, consistentes y precisos. Por una parte, es posible combinar sus especificaciones mediante la reprogramación, sin la adición de componentes.
Por otro lado, su carácter digital permite calcular y simular su respuesta usando procesadores de uso general (CPU o DSP), que tienen almacenado un algoritmo numérico para realizar el filtro.

Si se trata de un procesador digital de señal (DSP), este tiene la capacidad de realizar una suma de productos en un solo ciclo de reloj, teniendo por tanto una capacidad de procesamiento muy alta (Es multiinstrucción).

Fig. 1.3 Ejemplo de chip DSP

1.1.3. Respuesta de Impulso Infinita (IIR)

El filtro de Respuesta Infinita al Impulso o filtro IIR (por sus siglas en inglés Infinite Impulse Response), es un filtro digital en el que, si la entrada es una señal impulso, a la salida se tendrá un número infinito de términos no nulos.
Además, su salida depende no solo en la corriente y valores de entrada pasados, sino que también en los valores de salida pasados

1.1.4. Respuesta de Impulso Finita (FIR)

Del acrónimo en inglés Finite Impulse Response, es un filtro digital en el cual, si a la entrada se tiene una señal impulso, en la salida se tendrá un número finito de términos no nulos; la salida se basa en entradas actuales y anteriores.

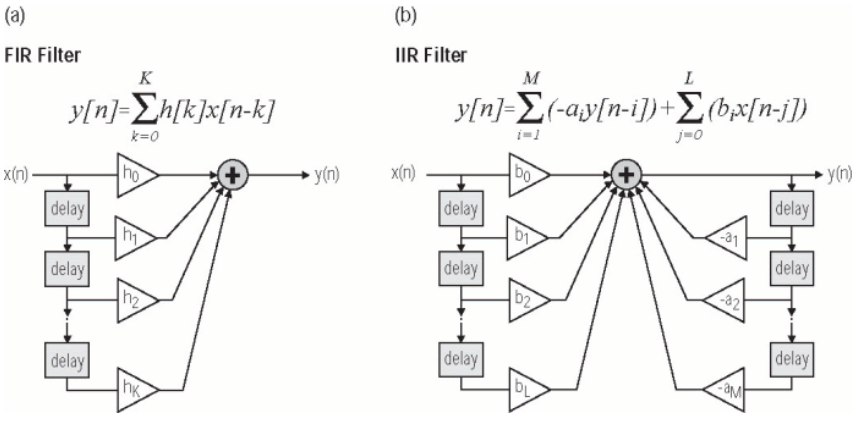


Fig. 1.4 Ecuación y estructura de los filtros FIR e IIR

1.2. Filtro Adaptativo

Los filtros adaptativos son sistemas que varían en el tiempo, de forma que se adaptan a cambios en su entorno, optimizando su funcionamiento de acuerdo a una serie de algoritmos conocidos como algoritmos adaptativos.
La forma de determinar el comportamiento óptimo del filtro adaptativo es minimizando una función monótona creciente de la señal o secuencia de error.
Esta secuencia se define como la diferencia entre una señal que se toma como referencia, o señal deseada, y la salida del filtro adaptativo.

En otras palabras, diseñar un filtro adaptativo consistirá en determinar la regla de variación de los coeficientes. El resto será automático.

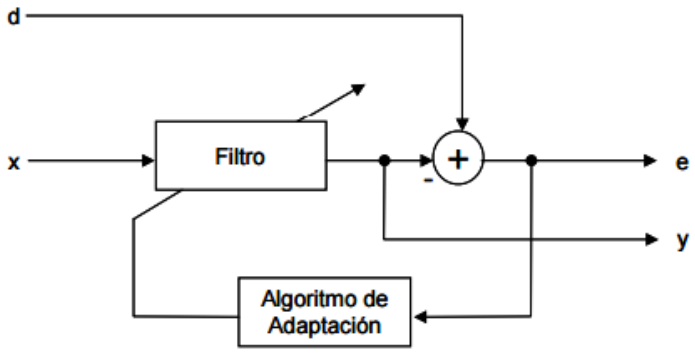


Fig. 1.5 Estructura General de un Filtro Adaptativo

Un filtro adaptativo es aquel cuyos coeficientes son actualizados mediante un algoritmo que cumple con un criterio predefinido, que puede ser minimizar el error cuadrático medio, como es el caso del ***LMS***.

Como consecuencia de que estos filtros no sean invariantes temporales y que no sean lineales, hace que su estudio sea más complejo que el de cualquier otro filtro digital, ya que no se pueden aplicar, salvo en un par de excepciones, las transformaciones en frecuencia, dominio Z, etc.

Se diferencia de los filtros digitales comunes tipo IIR o FIR, en que éstos tienen coeficientes invariantes en el tiempo.

1.3. Introducción al algoritmo LMS

En este apartado se describe un algoritmo ampliamente utilizado, el cual se conoce con el nombre de algoritmo de Mínimos Cuadrados Medios (LMS, por sus siglas en inglés).

Este algoritmo fue creado por Widrow y Hoff en el año de 1960 y pertenece a la familia de algoritmos de gradiente estocástico.

El término “gradiente estocástico” pretende distinguir el algoritmo LMS del método de máxima pendiente, que usa un gradiente determinista en un cálculo recursivo de los filtros Wiener para entradas estocásticas.

Una característica muy importante y a la vez atractiva del algoritmo LMS es su simplicidad. No requiere medición de funciones de correlación, tampoco necesita de inversión de matrices.

Es un algoritmo tan simple que lo utilizan como el estándar, con respecto a los otros algoritmos de filtrado adaptativo que existen.

Los más utilizados son este y sus múltiples variantes como pueden ser el Normalized LMS, el Proportionate Normalized LMS, el Afifine Projection Algorith y el Sign Algorithm (también llamado pilot LMS).

Estos se caracterizan en que se normaliza una parte del algoritmo, disminuyendo el error obtenido y consiguiendo una convergencia mucho más rápida que el LMS.
La parte negativa es que se vuelven mucho más complejos computacionalmente por lo que sigue saliendo a cuenta utilizar el LMS.

1.4. Descripción del LMS

El algoritmo LMS es un algoritmo de gradiente estocástico en el que se hacen iteraciones de cada valor de los coeficientes de un filtro transversal en la dirección del gradiente con la magnitud al cuadrado de la señal de error con respecto al valor del coeficiente.

Así que, el algoritmo LMS está estrechamente relacionado con el concepto de aproximación estocástica desarrollada por Robbins y Monro (1951) que se encuentra en estadística para resolver ciertos problemas de estimación de parámetros secuenciales.

La primera diferencia entre ellos, es que el algoritmo LMS utiliza un parámetro de tamaño de paso fijo “μ” para controlar la corrección aplicada a cada valor de los coeficientes de una iteración a la próxima, mientras que en el método de aproximación estocástica el parámetro del tamaño de paso es inversamente proporcional al tiempo o la potencia de n (n es la variable independiente de tiempo discreto).

Este algoritmo se compone de dos procesos básicos: Uno de filtraje y uno adaptativo.

- El primero involucra la salida procesada de un filtro lineal (transversal) en respuesta a una señal de entrada y la generación de una estimación de error resultante de la comparación de la salida con la respuesta deseada.

- El segundo implica el ajuste automático de parámetros del filtro de acuerdo con la estimación del error.

Así, que la combinación de estos dos procesos constituye un lazo de realimentación alrededor del algoritmo LMS.

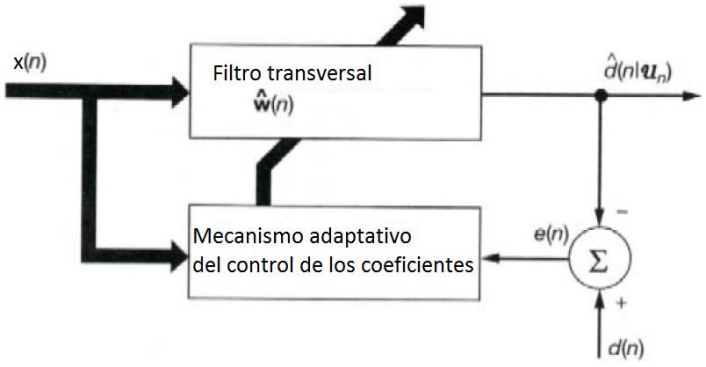


Fig. 1.6 Diagrama del bloque de la estructura LMS

Observando la Figura, tenemos un filtro transversal, alrededor del cual se construye el algoritmo LMS, el cual es responsable de desempeñar el proceso de filtrado.
Luego se tiene un mecanismo que desempeña el proceso de control adaptativo de los valores de los coeficientes del filtro transversal.

Los detalles de los componentes del filtro transversal son los siguientes:

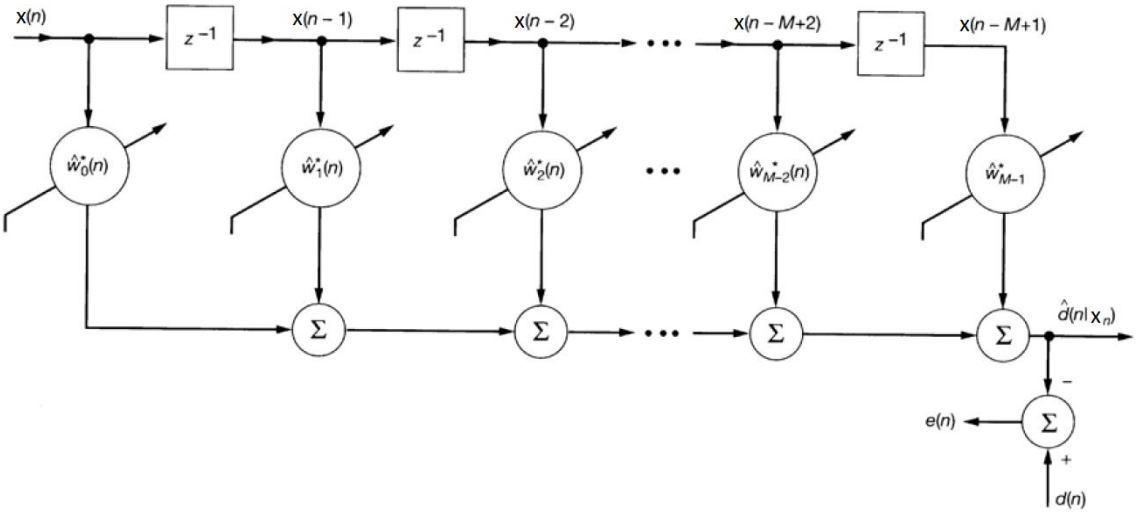


Fig. 1.7 Diagrama del filtro transversal

El valor calculado para el vector de coeficientes "$\hat{\mathbf{w}}(n)$" utilizando el algoritmo LMS representa una estimación del valor que se espera se aproxime a la solución de Wiener "$\hat{\mathbf{w}}_0$" para un entorno estacionario (esto es óptimo) con un número de n'' iteraciones que se acercan al infinito.

Luego, durante el proceso de filtrado la respuesta deseada "*d(n'')*" es suministrada para procesarla, al lado del vector de entrada "*x(n)".*

Obtenida esta entrada, el filtro transversal produce una salida "$\hat{d}(n/x_n)$" utilizada como una estimación de la respuesta deseada "*d(n).*"

Por consiguiente, se define una estimación de error "e(n)" como la diferencia entre la respuesta deseada y la salida del filtro.

La estimación del error "*e(n)"* y los valores del vector de entrada "*X(n)"* son aplicados al mecanismo de control y el lazo de realimentación alrededor de los valores de los coeficientes es cerrado consecuentemente.

La siguiente figura presenta los detalles del mecanismo de control adaptativo para los coeficientes, específicamente, una versión escalar del producto interno de la estimación del error “e(n)” y la entrada “x(n-k)” es calculada para k=0,1, 2, ..., M-2, M-1.

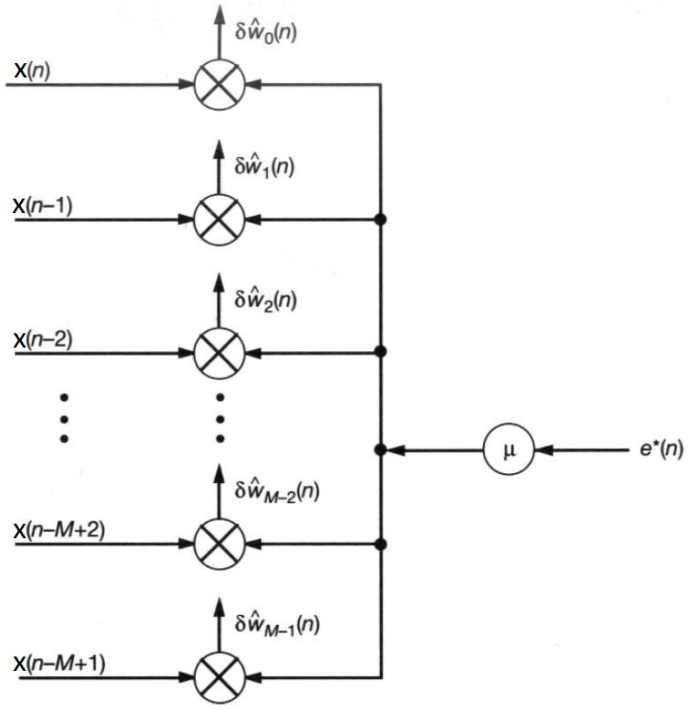


Fig. 1.8 Mecanismo de control adaptativo para los coeficientes

El resultado así obtenido define la correlación “$\delta\hat{w}_k(n)$” aplicada a los valores de los coeficientes “$\hat{w}_k(n)$” en la iteración n+1.

El factor de escala utilizado en este cálculo se designa por “μ” y es conocido como parámetro del tamaño de paso de adaptación o velocidad de adaptación

1.5. Ecuación del algoritmo LMS

El algoritmo LMS, para un filtro de orden "***M***", puede resumirse de la siguiente manera:

Parámetros:

M = Orden del filtro μ = Tamaño del paso

Inicialización: (Por defecto "0" si no hay datos del vector de coeficientes del filtro)

$$\hat{w}(0) = 0 \tag{1.1}$$

Datos:

Señal de entrada en el instante "n":

$$x(n) = [x(n), x(n-1), \ldots, x(n-M+1)]^T \tag{1.2}$$

$d(n)$: Señal deseada a la salida del filtro.

A calcular:

Estimación del vector de coeficientes (pesos) del filtro en el instante "n+1"

$$\hat{w}(n+1) = [\hat{w}_0(n+1), \hat{w}_1(n+1), \ldots, \hat{w}_{M-1}(n+1)]^T \tag{1.3}$$

Cómputo:

Para n=0, 1, 2, ..., calcular:

Señal de error:

$$e(n) = d(n) - \hat{w}^H(n) \cdot x(n) \tag{1.4}$$

Adaptación de los coeficientes del filtro:

$$\hat{w}(n+1) = \hat{w}(n) + \mu\, e^*(n) x(n) \tag{1.5}$$

Salida del filtro:

$$y(n) = w^H(n) \cdot x(n) \tag{1.6}$$

"$\_^{T}$" denota trasposición, "$\_^{H}$" denota traspuesta conjugada, el asterisco denota conjugación.

El factor “μ” que determina la velocidad de convergencia, es directamente proporcional a la velocidad de convergencia e inversamente proporcional al error cuadrático medio mínimo.
Por lo tanto, existe un compromiso entre velocidad de convergencia y error cuadrático promedio mínimo, es en este punto donde radica este algoritmo presenta su mayor inconveniente.

CAPÍTULO 2. Uso de la GPGPU para la paralelización

2.1. Uso de la GPU ante el DSP

El procesado de señales multidimensionales es un problema común en la investigación científica. Normalmente, la complejidad de cálculo de un problema de DSP crece exponencialmente con el número de dimensiones.
Sin embargo, con un alto grado de complejidad de almacenamiento y tiempo, es extremadamente difícil procesar señales multidimensionales en tiempo real.

Aunque se han propuesto muchos algoritmos rápidos (por ejemplo, FFT) para problemas de DSP 1-D (vectores de 1 dimensión), todavía no son lo suficientemente eficaces para adaptarse a problemas con mayor número de dimensiones.

Todavía es difícil obtener los resultados de cálculo deseados con procesadores de señales digitales (DSPs). Es por eso que se necesitan mejores algoritmos y arquitectura de hardware para acelerar los cálculos DSP multidimensionales.

Las unidades de procesamiento gráfico de propósito general (GPGPU) modernas se considera que tienen un rendimiento excelente en operaciones vectoriales y manipulaciones numéricas a través de un gran número de cálculos paralelos.
El término GPGPU es concepto reciente dentro de informática que trata de estudiar y aprovechar las capacidades de cómputo de una GPU.

Mientras que el procesamiento de señales digitales, en particular las señales multidimensionales, a menudo implica una serie de operaciones de vector en una gran cantidad de muestras de datos independientes, las GPGPUs ahora son comúnmente utilizadas para acelerar los DSP multidimensionales, tales como el procesamiento de imágenes, señales, códecs de vídeo y ultrasonido.

Conceptualmente, el uso de dispositivos GPGPU, para realizar las tareas que haría el DSP multidimensional, es capaz de reducir drásticamente la complejidad computacional en comparación con las unidades centrales de procesamiento (CPUs), procesadores de señales digitales (DSP) u otros aceleradores FPGA.

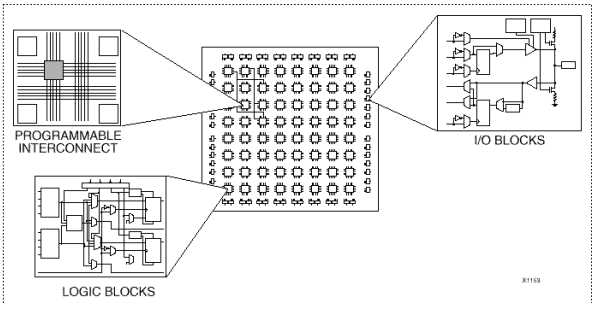


Fig. 2.1 Arquitectura interna de una FPGA

2.2. Ventajas del uso de la GPU

La demanda en el mercado de consumo, de hardware gráfico que acelera el renderizado de imágenes en 3D, ha dado lugar a tarjetas gráficas que son capaces de ofrecer sorprendentes niveles de rendimiento.

Estos resultados se lograron adaptando específicamente el hardware para esa finalidad. Los aceleradores gráficos están volviéndose cada vez más programables por lo que este rendimiento los ha convertido en un objetivo atractivo para otros dominios.

Las Unidades de Procesamiento Gráfico (GPU) proporcionan una arquitectura de computación paralela de bajo costo.

Es posible lograr un paralelismo masivo con la técnica del SIMT (Single Instruction Multiple Thread) en la Unidad de Procesamiento de Gráficos de Uso General (GPGPU) integrada con la Unidad Central de Procesamiento (CPU).

Las ventajas de utilizar SIMT (Instrucción Única para Múltiples Hilos) pueden explicarse mediante un ejemplo: cambiando el brillo de una imagen.
Cada píxel de una imagen consta de tres valores para el brillo siendo los colores: rojo (R), verde (G) y azul (B).
Normalmente se cambiaría el valor de cada pixel secuencialmente, lo que duraría un cierto tiempo.

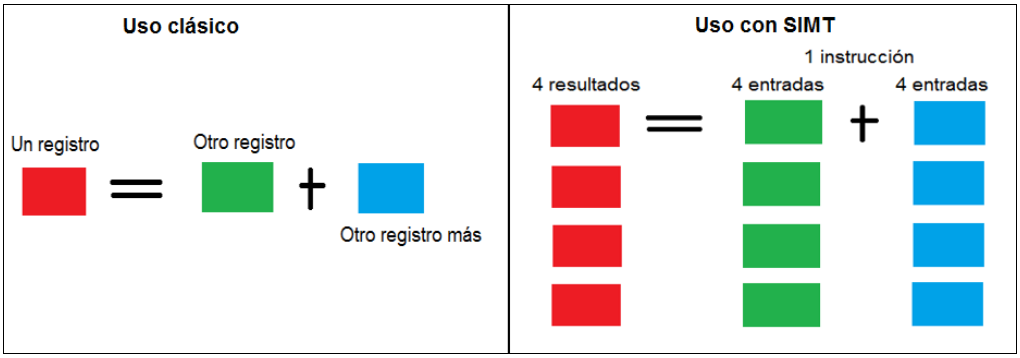


Fig. 2.2 Instrucciones de forma clásica e Instrucciones utilizando SIMT

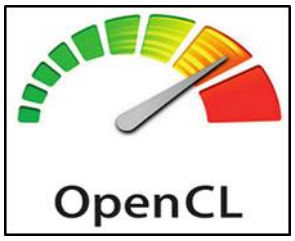
Con un procesador que utilice SIMT hay dos mejoras en este proceso. Primeramente, los datos se organizan en bloques, y una serie de valores se pueden ser cargados de una vez.

En lugar de usar una serie de instrucciones que digan: "recuperar este píxel, ahora recuperar el píxel siguiente, etc", un procesador SIMT tendrá una sola instrucción que efectivamente dirá: "recuperar n píxeles" (donde n es un número que varía de diseño a diseño).
Por un gran número de razones, esto puede tardar mucho menos tiempo que recuperar cada píxel individualmente, como resultaría con el diseño tradicional de la CPU.

2.3. GPU – CUDA vs OpenCL

Los archivos con gran número de datos consumen una enorme cantidad de tiempo en ser procesados por la CPU.
Se requiere mucho tiempo para realizar muchas operaciones en ella, lo que resulta en la disminución del rendimiento.

CUDA es un modelo de programación paralela por el cual se mejora el rendimiento en términos de tiempo para cualquier tipo de computación.



Su alternativa, OpenCL (Open Computing Language), se diferencia pues es un estándar de programación abierto lo que le permite ser usado en cualquier GPU.
AMD es su patrocinador y lo que pretende es que no sea necesario utilizar exclusivamente dispositivo NVIDIA por lo que soporta varias clases de dispositivos como pueden ser: CPUs, GPUs, DSPs, teléfonos móviles, etc.

Para trabajar con CUDA, es tan sencillo como descargarse desde su misma web, un paquete que incluye el kit de herramientas CUDA, controladores de desarrollador y el SDK.
En cambio, para OpenCL, las herramientas necesarias y el SDK son proporcionados por cada proveedor de sus dispositivos compatibles.

OpenCL consta de una serie de herramientas y una API (Interfaz de programación de aplicaciones) a diferencia de CUDA que representa toda una arquitectura como veremos más adelante.

A día de hoy CUDA sigue ofreciendo mayor rendimiento por varios motivos:

- Puede usarse más de una GPU para GPGPU.
- CUDA es compatible con más aplicaciones (Adobe CC, Premiere Pro, etc).
- Mayor optimización pues está ejecutándose en su propio hardware.

La forma más eficiente de trabajo es el uso conjunto de CPU y GPU mediante la arquitectura de cálculo paralelo CUDA.

Por ello, más adelante, veremos la diferencia entre el procesado en CPU y en GPU, que reside principalmente en el hardware utilizado y en la forma que cada núcleo procesa los datos.

2.4. Aplicaciones de cálculo en la GPU (CUDA)

Desde que debutó, diversidad de empresas y aplicaciones han disfrutado mucho de un gran éxito por haber escogido CUDA C como plataforma para crear sus aplicaciones.

Estos beneficios a menudo incluyen aumentos significativos de rendimiento en los últimos pasos de la implementación de la técnica. Además, las aplicaciones que se ejecutan sobre procesadores gráficos NVidia, gozan de un rendimiento superior por dólar y por vatios, que las implementaciones construidas exclusivamente para tecnologías de CPU.

Los sistemas de radar suelen requerir reconstruir una cantidad numerosa de muestras de datos en 3-D o 4-D en tiempo real. Tradicionalmente y especialmente en el campo militar, esto necesita el apoyo de los superordenadores.

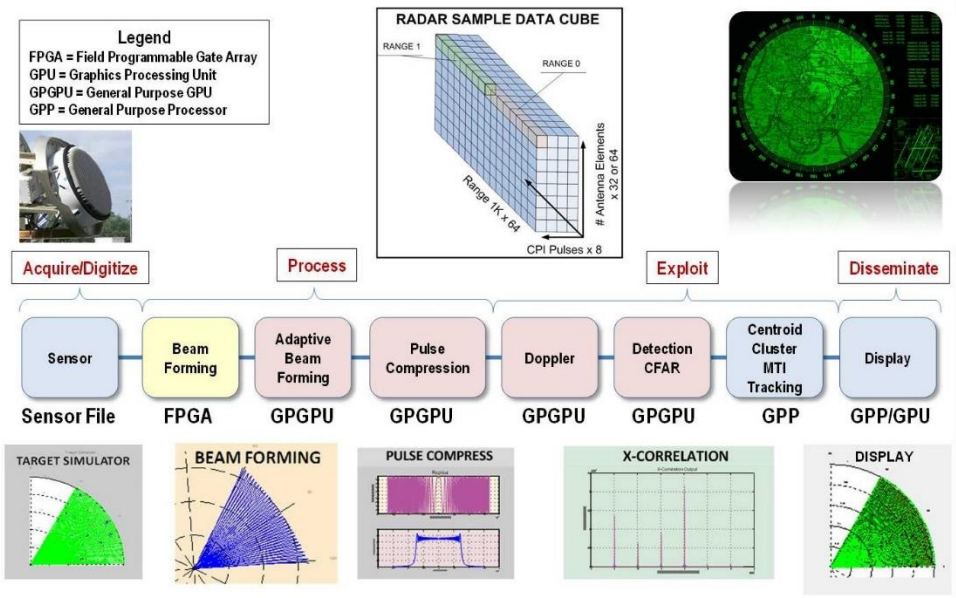


Fig. 2.3 Procesado GPU en sistemas radar

Hoy en día, las GPGPUs se emplean para reemplazar superordenadores para procesar señales de radar. Por ejemplo, para señales de radar de apertura sintética (SAR), que normalmente implica cálculos multidimensionales de FFT.

Los GPGPUs se pueden utilizar para realizar rápidamente FFT y / o iFFT en este tipo de aplicaciones.

Muchos automóviles de conducción automática aplican técnicas de reconocimiento de imagen 3D para controlar automáticamente los vehículos. Es evidente que, para adaptarse al entorno exterior que cambia rápidamente, los procesos de reconocimiento y decisión deben realizarse en tiempo real.

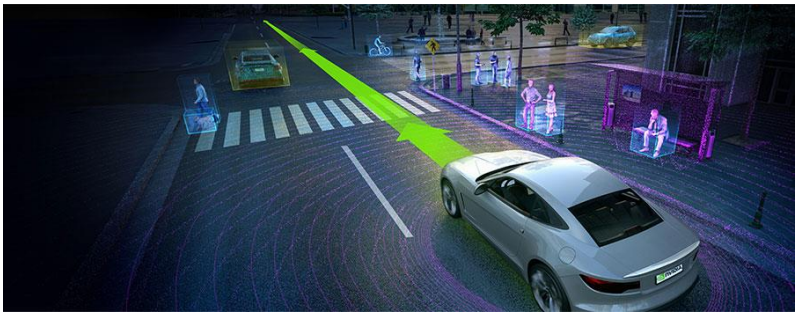


Fig. 2.4 Ejemplo de coche de conducción automática

Otros usos de las GPU (CUDA) pueden ser:

• Bioinformática
Para la secuencia y el acoplamiento de proteínas se requieren procesos informáticos muy intensos.

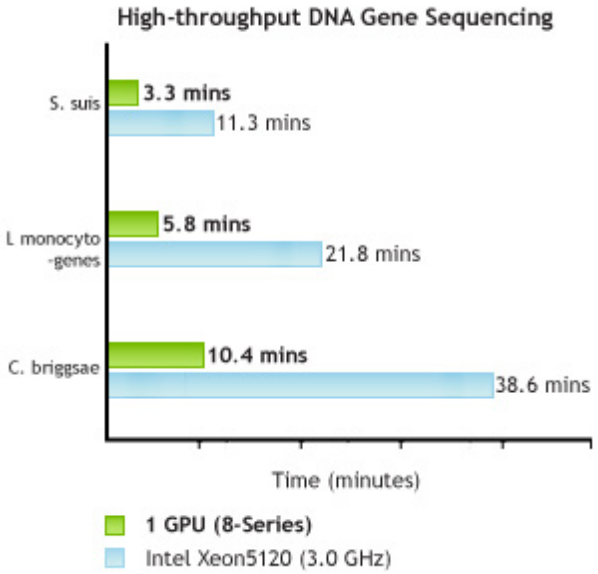


Fig. 2.5 Comparación GPU-CPU en el alineamiento de secuencias de ADN

• Ciencia de los datos, analítica y bases de datos
Se usa para analizar cantidades masivas de datos (Big Data) y mejorar así la toma de decisiones de negocio en tiempo real.

• Defensa e Inteligencia
Hay más de 100 millones de huellas dactilares y fotografías almacenadas en la base de datos del FBI. Las GPUs aceleran todas las funciones que intervienen en el procesamiento de esas imágenes, lo que incluye la georrectificación, los algoritmos de filtrado, la detección de cambios y la reconstrucción en 3D.

• Cálculo financiero
La enorme aceleración que proporcionan las GPUs de NVIDIA a aplicaciones de cálculo tan utilizadas como las simulaciones Montecarlo representa una gran ventaja competitiva para las empresas de servicios financieros.

Poder calcular el precio y el riesgo de opciones complejas y derivados OTC (mercados donde se negocian acciones, bonos, etc.) en segundos en lugar de horas permite ejecutar más simulaciones y, por tanto, mejorar la calidad y fiabilidad de los resultados

• Aprendizaje automático
Es una rama cada vez más importante de la inteligencia artificial (IA) y es la ciencia que permite a los ordenadores actuar sin haber sido expresamente programados para ello. (Machine Learning).

Las redes neuronales basadas en sistemas informáticos pueden aprender a imitar el comportamiento del cerebro, lo que incluye las capacidades que éste posee para reconocer objetos, personas, voces y sonidos.

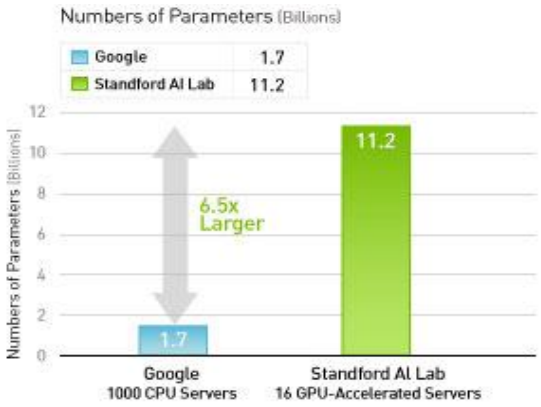


Fig. 2.6 Comparación entre servidores CPU y GPU

NVIDIA ha colaborado con un equipo de investigadores de la Universidad de Stanford para crear la red neuronal artificial más grande del mundo con el objetivo de reproducir la forma en que aprende la mente humana.
La red es 6,5 veces más grande que la anterior poseedora del récord, desarrollada por Google en 2012

Entre otras muchas aplicaciones, estas son las que más utilidad están aportando:
• Dinámica de fluidos computacional (CFD), Química cuántica, Exploración sísmica, Modelos meteorológicos y climáticos, procesado de señales.

2.5. Paralelismo eficiente

Una de las ventajas de los FPGAs (Field Programmable Gate Array) y GPUs es el paralelismo arquitectónico que contienen estos dispositivos.
El problema, es que normalmente tienen frecuencias de reloj más bajas que las CPUs convencionales. Esto significa que, para lograr una aceleración, el paralelismo del dispositivo debe ser lo más eficiente posible, identificando y solventando el problema a resolver.

Teóricamente, si uno dobla el número de procesadores, el tiempo de ejecución debería reducirse a la mitad, pero no es así y veremos el porqué:

La ley de Amdahl es un modelo matemático que describe la relación entre la aceleración esperada de la implementación paralela de un algoritmo y la implementación serial del mismo algoritmo.

Supongamos que una porción P de un cálculo puede ser paralelizada y que existen N rutas de datos paralelas disponibles (número de procesadores).
La ley de Amdahl establece que la máxima velocidad alcanzable a través de la paralelización es: $S = \frac{1}{(1-P)+\left(\frac{P}{N}\right)}$

Cada vez que se dobla el número de procesadores la aceleración disminuye, de esta manera se tiende al límite siguiente: $\frac{1}{1-p}$

Si el 95% de un programa es paralelizable, la máxima aceleración obtenida será de 20x como vemos en el siguiente gráfico:

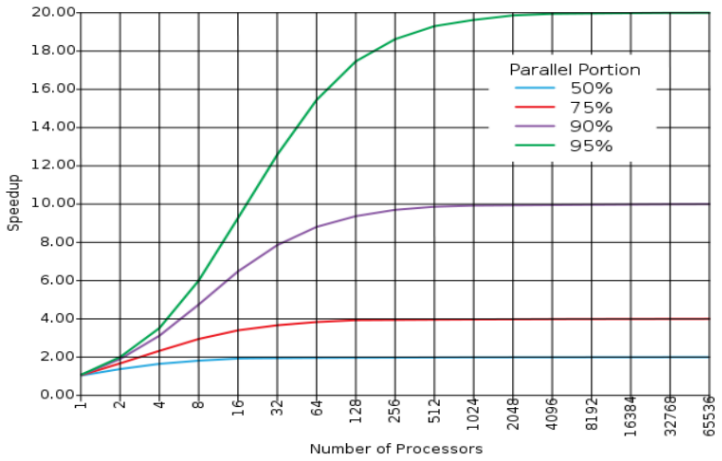


Fig. 2.7 Gráfico de la ley de Amdahl sobre la paralelización

Es importante señalar que la ley de Amdahl proporciona una aceleración máxima alcanzable, no la aceleración que se obtendrá.
La aceleración real puede estar limitada por el hardware utilizado y el tiempo de comunicación entre la CPU y la GPU.

2.6. Comparación GPU-CPU

Las GPU que están disponibles hoy en día proporcionan una alta potencia de cálculo a bajo costo y han sido descritas como superordenadores de sobremesa.
Son capaces de realizar operaciones masivamente paralelas usando hilos CUDA. Las GPU se han utilizado para muchos cálculos de propósito general debido a su bajo costo, alta potencia de cálculo y alta disponibilidad.

Fig. 2.8 GPU con tecnología CUDA

CUDA de NVIDIA presenta un modelo de programación heterogéneo donde el hardware paralelo se puede utilizar junto con la CPU.

Los modelos de programación GPU están adaptados de tal manera que el compilador puede razonar sobre la aplicación y extraer el paralelismo automáticamente.
Ejemplos de esto pueden ser DirectX, CUDA y Cg (lenguaje de alto nivel para la programación de píxel shaders).

La arquitectura Intel tiene un propósito más general que la arquitectura de GPU y otros coprocesadores.

A diferencia de las GPU, las arquitecturas CPU de Intel disponen de:

1) Comunicación entre núcleos a través de jerarquías de caché sustanciales y coherentes.
2) Sincronización de hilos de latencia baja en toda la matriz del procesador.
3) Ancho SIMD más estrecho y eficaz. (Single Instruction Multiple Data).

En lenguaje de alto nivel, el objetivo es definir un modelo de programación restringido que sea eficiente y se dirija a los núcleos altamente paralelos, como son los “Intel multi-core” y los sistemas “Intel Tera-Scale” (Estos últimos pertenecen a un programa de investigación que busca aumentar de forma considerable el número de núcleos por CPU, que ronda los 80 actualmente).

Hay diferentes maneras de clasificar los tipos de ordenadores.
Una de las clasificaciones más utilizadas, en uso desde 1966, se llama "Taxonomía de Flynn".

Las cuatro clasificaciones definidas se basan en el número de instrucciones concurrentes (control) y en los flujos de datos disponibles en la arquitectura:

- Una instrucción, un dato (**SISD**).
    - Ordenador secuencial – CPU.
- Múltiples instrucciones, un dato (**MISD**).
    - No se usa pues se necesitarían varios flujos de datos.
- Una instrucción, múltiples datos (**SIMD**)
    - Computación paralela – GPGPU.
- Múltiples instrucciones, múltiples datos (**MIMD**)
    - Computación distribuida – Clústeres de ordenadores.

Los procesadores basados en GPU pueden realizar eficientemente operaciones de coma flotante y usar paralelismo a niveles masivos por lo que pueden ser una opción adecuada para procesar grandes cantidades de datos.

El cálculo acelerado en la GPU traslada las partes de la aplicación con mayor carga computacional a la GPU y deja el resto del código ejecutándose en la CPU. Desde la perspectiva del usuario, las aplicaciones simplemente se ejecutan más rápido.

Una forma sencilla de entender la diferencia entre la GPU y la CPU es comparar la forma en que procesan las tareas. Una CPU está formada por varios núcleos optimizados para el procesamiento en serie, mientras que una GPU consta de millares de núcleos más pequeños y eficientes diseñados para manejar múltiples tareas simultáneamente.

Las GPU actuales poseen hasta casi 4000 núcleos que procesan las cargas de trabajo de forma paralela y muy eficiente.

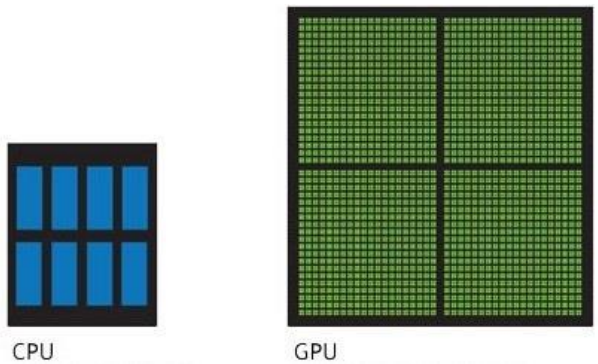


Fig. 2.9 Cantidad de núcleos de las CPU y GPU

2.7. Características del Hardware

Impulsado por la gran demanda del mercado de gráficos 3D de alta definición, la GPU ha evolucionado hasta convertirse en un procesador de núcleo muy paralelo, multiproceso, con una gran potencia computacional y un ancho de banda de memoria muy alto.

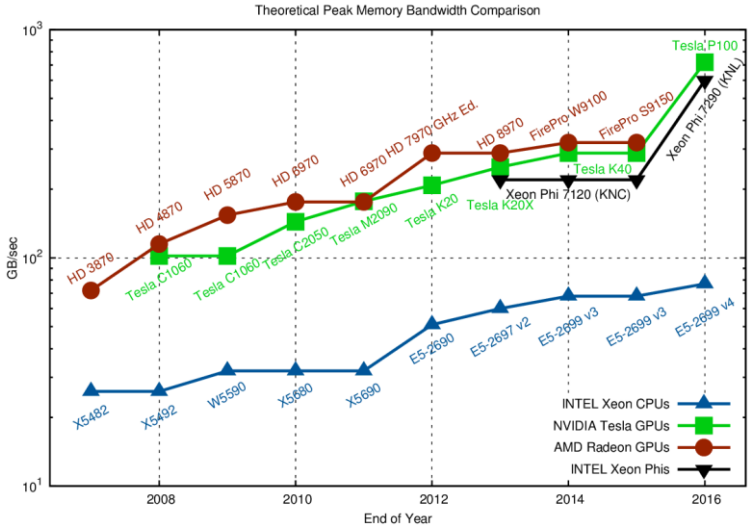


Fig. 2.10 Evolución anual del ancho de banda de las GPU

En informática, las operaciones de coma flotante por segundo son una medida del rendimiento de un ordenador, especialmente en cálculos científicos que requieren un gran uso de operaciones de coma flotante.
Es más conocido su acrónimo, FLOPS (del inglés floating point operations per second).

En la siguiente imagen puede apreciarse como en pocos años, la capacidad de rendimiento de los procesadores (CPU y GPU) ha aumentado enormemente.

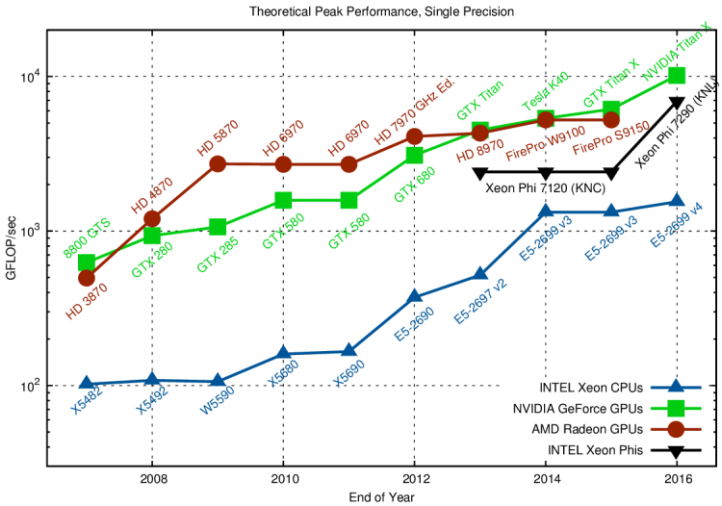


Fig. 2.11 Evolución anual del rendimiento de las GPU

El valor más alto actualmente, está en **12 TFLOPs** para la nueva tarjeta gráfica “NVIDIA Titan Xp”, que contiene nada menos que 3840 núcleos CUDA.

La “capacidad de coma flotante” difiere entre la CPU y la GPU pues esta última está especializada en computación intensiva.

Su “fuerte” reside en la capacidad de realizar cálculos altamente paralelos (exactamente lo que representa el procesamiento de gráficos) por lo que se ha diseñado para que haya más transistores dedicándose al procesamiento de datos que al almacenamiento en cache de datos (como en la CPU).

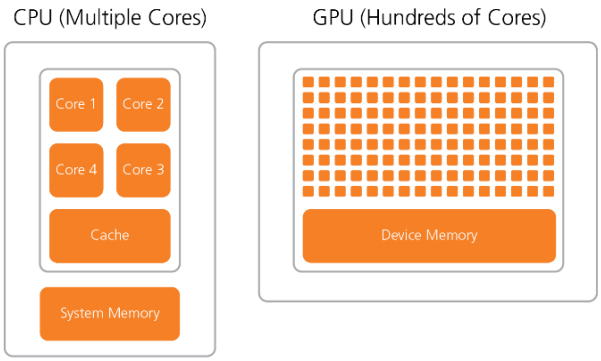


Fig. 2.12 Arquitectura interna de CPU y GPU

CUDA viene con un entorno de software que permite a los desarrolladores usar C como un lenguaje de programación de alto nivel.
Como se ilustra en la siguiente imagen, se admiten otros lenguajes o APIs como FORTRAN, DirectCompute, OpenACC.

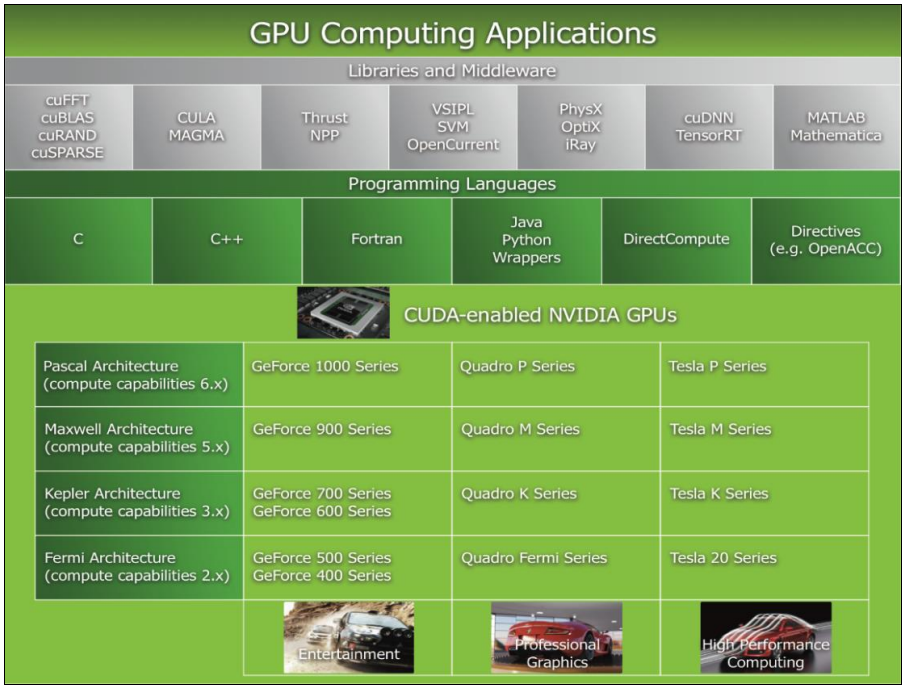


Fig. 2.13 Aplicaciones que usan GPU

CAPÍTULO 3. CUDA y paralelización del algoritmo LMS

3.1. Historia de CUDA

Mientras que la investigación sobre el uso de procesadores para la computación general lleva décadas en curso, la creación de aceleradores modernos para el uso en computación de alto rendimiento (HPC) no ocurrió hasta finales de 2000.

En esa época, los procesadores especializados llamados unidades de procesamiento gráfico (GPU), producidos por NVIDIA y ATI / AMD, se estaban fabricando con capacidades suficientes para soportar las cargas de trabajo HPC.

Durante este tiempo, NVIDIA lanzó la primera beta pública de su “CUDA Framework” (2007), así como la primera generación de sus GPUs “Tesla” comercializados en el sector de la HPC.
Hoy en día es muy común utilizar estas GPGPU (Unidades de Procesamiento Gráfico de Propósito General) para cualquier tipo de trabajo relacionado con HPC.

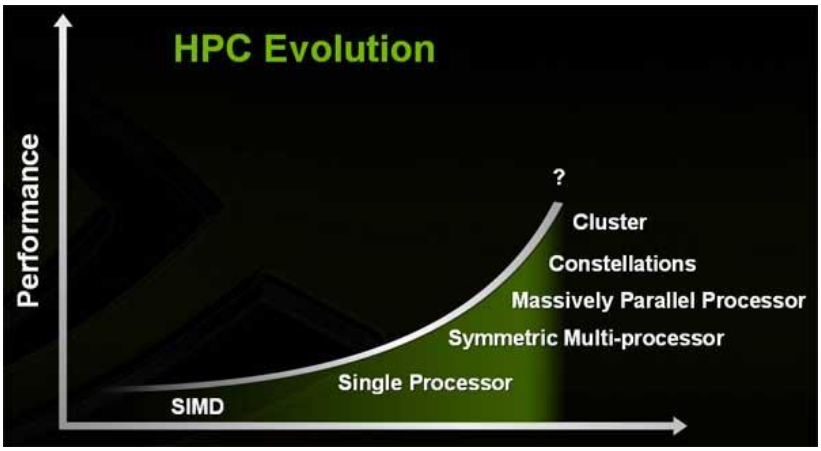


Fig. 3.1 Evolución de la HPC gracias a las GPGPU

“CUDA Framework” se utiliza en las GPU NVIDIA y su competidor, “OpenCL Framework", se utiliza tanto en las GPUs AMD como en las GPU NVIDIA.
Otras técnicas y herramientas de cálculo acelerado se han construido sobre estos dos “GPU Frameworks” para ayudar a los programadores a acelerar aplicaciones con tecnologías de GPU.

Mientras que otras tecnologías de aceleración están emergiendo, las GPUs son actualmente la tecnología de acelerador más ampliamente adoptada en la High performance Computing.

3.2. Ventajas de CUDA

CUDA se dirige tanto a educación como a investigación.

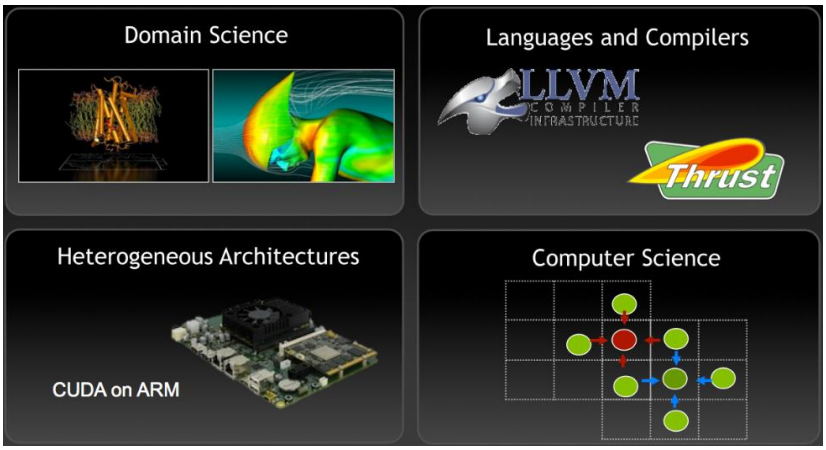


Fig. 3.2 Dominios de uso de CUDA

Soporta un ecosistema muy diverso en torno a la programación paralela.
Este ecosistema puede articularse en 4 vertientes principales:

- La cadena de herramientas de compilación.
- Los lenguajes de programación.
- Las librerías de código.
- Las herramientas de desarrollo.

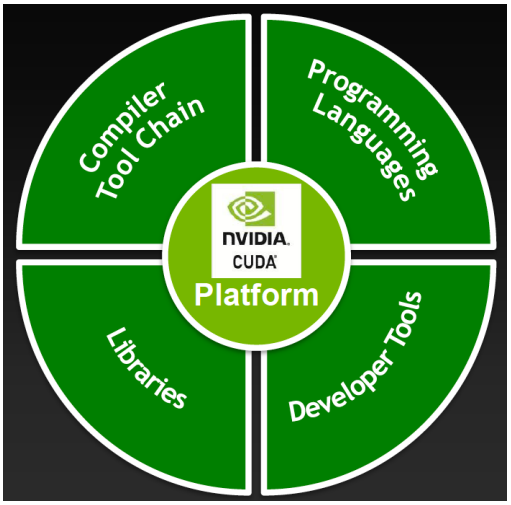


Fig. 3.3 Dominios de uso de CUDA

El uso de las librerías que incorpora simplifica enormemente el trabajo a la vez que acelera muchas de las funciones que comúnmente utilizamos.
Las han desarrollado expertos de Nvidia por lo que aseguran un rendimiento óptimo. También hay muchos programadores aficionados intentando exprimir al máximo esta gran tecnología.

3.3. Modelo de programación

El modelo de programación CUDA asume que los hilos (threads) CUDA se ejecutan en una unidad física distinta que actúa como coprocesador (device o GPU) al procesador (host o CPU) donde se ejecuta el programa.

CUDA C es una extensión del lenguaje de programación C, que permite al programador definir funciones C, llamadas kernels, que, al ser llamadas, se ejecutan en paralelo por N hilos diferentes. Los kernels se ejecutan de forma concurrente en el device.

Como ejemplo, el siguiente código muestra cómo se define un kernel y cómo se llama desde un programa:

```
// Kernel definition
__global_- void VecAdd( f loat * A, f loat * B, f loat * C)
{
        int i = threadIdx . x;
        C[ i ] = A[ i ] + B[ i ];
}

int main ( )
{
       // Kernel invocation with N threads
       VecAdd<<<1, N>>>(A, B, C);
}
```

Existe una jerarquía perfectamente definida sobre los hilos de CUDA. Los hilos se agrupan en vectores a los que se les llama bloques, estos vectores pueden ser de una, dos o tres dimensiones, de forma que definen bloques de hilos de una, dos o tres dimensiones.

Hilos del mismo bloque pueden cooperar entre sí, compartiendo datos y sincronizando sus ejecuciones

Sin embargo, hilos de distintos bloques no pueden cooperar. Los bloques a su vez, se organizan en un grid de bloques. Este grid, de nuevo puede ser de una o dos dimensiones.

Los valores entre <<< ... >>> que aparecen en código anterior se conocen como la configuración del kernel, y definen la dimensión (1) del grid y el número de hilos de cada bloque (N).

La siguiente figura muestra cómo se organizan los hilos en un grid de 2x3 bloques y 3x4 hilos cada uno.

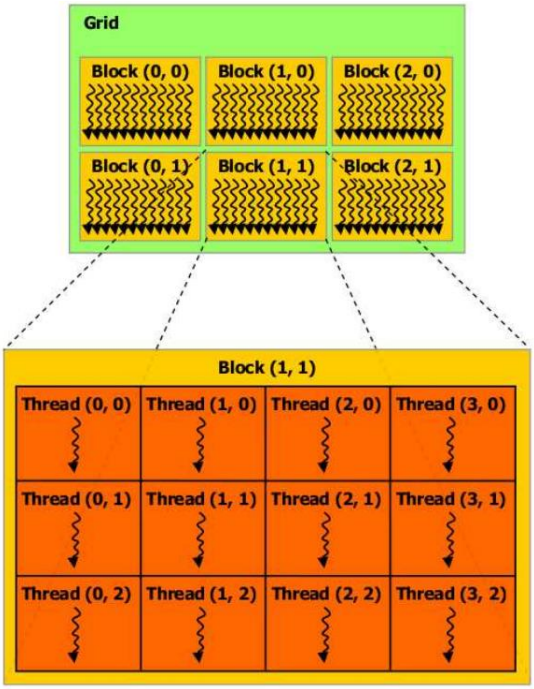


Fig. 3.4 Jerarquía de los hilos en la GPU

Como puede verse, cada hilo queda perfectamente identificado por un ID de bloque y el ID del propio hilo dentro del bloque. Estos IDs suelen usarse como índices para definir qué porciones de los datos procesa cada hilo.

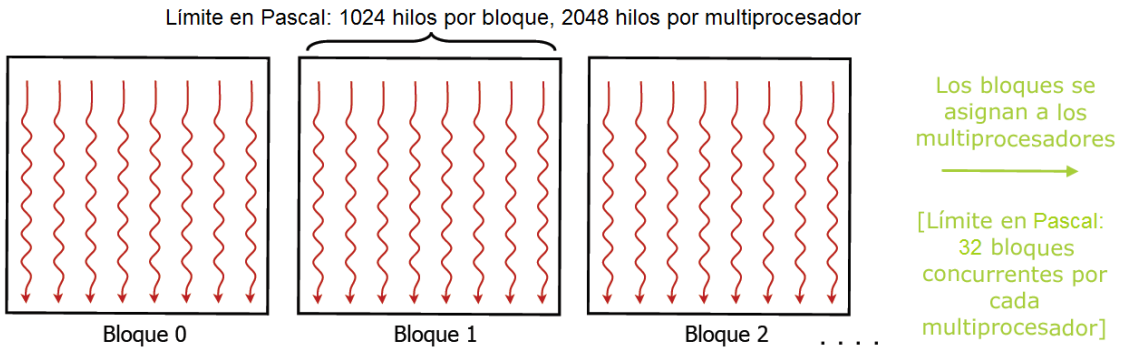


Fig. 3.5 Bloques e hilos en GPU

Los hilos de un bloque comparten la “shared memory” (memoria compartida) y pueden sincronizarse mediante llamadas a "synchthreads()".

Al asignar el tamaño de los hilos y bloques hay que tener en cuenta las restricciones del hardware que estamos utilizando pues cada GPU, dependiendo de su generación, posee un número máximo de bloques recurrentes.

La arquitectura Pascal, que actualmente es la última, está limitada a 1024 hilos por bloque y 32 bloques concurrentes.
Esto puede observarse si ejecutamos la función “gpuDevice” en Matlab lo que nos devolverá la información de la tarjeta gráfica que tengamos seleccionada.

```
>> gpuDevice

ans =

  CUDADevice with properties:

                      Name: 'GeForce GTX 970'
                     Index: 1
         ComputeCapability: '5.2'
            SupportsDouble: 1
             DriverVersion: 8
            ToolkitVersion: 7.5000
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
               MaxGridSize: [2.1475e+09 65535 65535]
                 SIMDWidth: 32
               TotalMemory: 4.2950e+09
           AvailableMemory: 3.4699e+09
      MultiprocessorCount: 13
             ClockRateKHz: 1329000
              ComputeMode: 'Default'
     GPUOverlapsTransfers: 1
   KernelExecutionTimeout: 1
         CanMapHostMemory: 1
          DeviceSupported: 1
           DeviceSelected: 1
```

En mi caso, uso una Nvidia GTX 970 que está enfocada a los videojuegos, pero que como es bastante nueva ofrecerá un buen rendimiento gracias al número de “MaxThreadsPerBlock”.

El gran avance se ha producido con la última arquitectura desarrollada por Nvidia, Pascal, que permite una concurrencia de 32 bloques y ofreciendo mayor potencia de procesado paralelo, a diferencia de la anterior que solo permitía 16 bloques concurrentes.

Es muy importante remarcar que a medida que aparecen nuevas tarjetas gráficas, el número de núcleos CUDA que contienen aumenta considerablemente.
El problema es que en cada generación de GPUs la arquitectura cambia, por lo que, aunque una nueva GTX Titan Xp (Pascal) posea 3840 CUDA Cores y la antigua GTX Titan Z (Kepler) posea 5760, el rendimiento se verá realmente afectado por la capacidad de procesado (FLOPS) en la que la nueva tarjeta es mucho superior a la Titan Z. (12 TFOPS contra 8 TFLOPS).

Las limitaciones al escoger los tamaños de hilo (thread) y bloque para un multiprocesador Pascal son:

1- 32 bloques concurrentes.
2- 1024 hilos/bloque.
3- 2048 hilos en total.

Ejemplos de limitación:

- 1 bloque de 2048 hilos
  - No lo permite [2].
- 2 bloques de 1024 hilos.
  - Posible en el mismo multiprocesador. (2048 hilos en total)
- 4 bloques de 512 hilos.
  - Posible en el mismo multiprocesador. (2048 hilos en total)
- 4 bloques de 1024 hilos.
  - No lo permite [3] en el mismo multiprocesador, pero es posible usando dos multiprocesadores. (4096 hilos en total)
- 8 bloques de 256 hilos.
  - Posible en el mismo multiprocesador. (2048 hilos en total)
- 256 bloques de 8 hilos.
  - No lo permite [1] en el mismo multiprocesador, pero es posible usando 8 multiprocesadores (8 * 32 bloques).

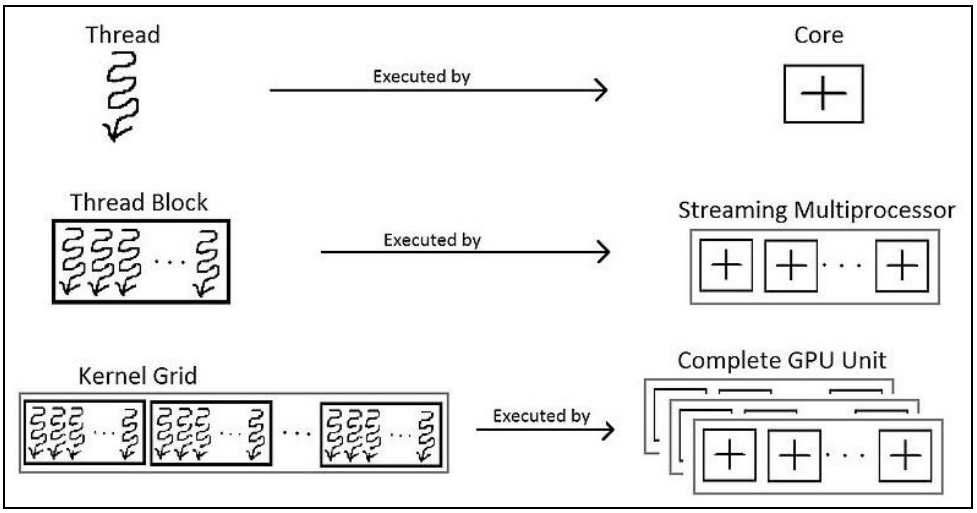


Fig. 3.6 Perspectiva del software para el bloque de hilos

3.4. Aceleración de MATLAB usando MEX

MATLAB proporciona un script, llamado mex, para compilar un archivo MEX a un objeto compartido o dll que se puede cargar y ejecutar dentro de un programa de MATLAB.

Este script es capaz de analizar el código C, C++ y FORTRAN y puede llamar al código CUDA para crear una mayor optimización en las GPUs.

A pesar de que MATLAB llama a bibliotecas optimizadas internamente, todavía hay espacio para una mayor optimización.

Los archivos MEX se han utilizado en el pasado como una forma de cargar bibliotecas multiproceso o vectorizadas y se usan en este caso para ejecutar código en la GPU y para manejar la transferencia de datos entre el host (CPU) y el device (GPU).

Todos los archivos MEX deben incluir 4 elementos:

1. "#include mex.h" (para C y C++)

2. La rutina "gateway" a cada archivo MEX se llama "mexFunction".
   Este es el punto de entrada que MATLAB utiliza para acceder a la DLL.

   En C/C++, siempre se usa de la siguiente manera:
   "MexFunction (int nlhs, mxArray * plhs [], int nrhs, const mxArray * prhs [])"

   Dónde:
   Nlhs = número de mxArrays esperados (Left Hand Side)
   Plhs = vector de punteros a la salida esperada
   Nrhs = número de entradas (Right Hand Side)
   Prhs = vector de punteros a los datos de entrada.
   (Los datos de entrada son de sólo lectura)

3. "mxArray":

   Es una estructura especial que contiene los datos de MATLAB. Es la representación C de un array MATLAB. Todos los tipos de matrices MATLAB (escalares, vectores, arrays, strings, etc.) son mxArrays.

```
void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    double *inMatrix1;              /* input matrix axBB*/
    double *inMatrix2;              /* input matrix ayBB*/
    double *outMatrix1;             /* output matrix ayBB_est*/
    double *outMatrix2;             /* output matrix error*/
    size_t nrows1;                  /* size of matrix inMatrix1*/
    size_t nrows2;                  /* size of matrix inMatrix2*/
    int ncol1;
    int ncol2;
```

4. Funciones API (como asignación de memoria y liberación de esta).

Para poder ejecutar este tipo de programas hay que escribir el siguiente comando en Matlab: “mex ejemplo.c”.

Este comando genera un archivo compilado MEX. (El sufijo producido depende del sistema operativo):

ejemplo.mexw32 (Windows 32 bit)
ejemplo.mexw64 (Windows 64 bit)
ejemplo.mexglx (Linux 32 bit)

Fig. 3.7 Carpeta tras compilación del código

Para que MATLAB pueda ejecutar sus funciones en C, hay que poner los archivos compilados MEX que contienen esas funciones en un directorio en la ruta MATLAB o ejecutar MATLAB en el directorio en el que se hallan.

Para la mayoría de los casos, el archivo MEX contiene código CUDA (funciones del kernel, configuración) y para ser procesado por el compilador CUDA, el archivo necesita tener un sufijo ".cu".

Es muy parecido al anterior ejemplo de “mex ejemplo.c”, pero en este caso se utiliza “mexcuda ejemplo.cu” para poder ejecutar con MATLAB todo el código que contenga CUDA.

Los archivos CUDA MEX se parecen a los MEX (C, C++ y FORTRAN) pero difieren a la hora de asignar los datos en la memoria pues hay que hacerlo en la GPU y también hay que realizar los procesos que se encuentran entre CPU-GPU, como es la llamada a los kernels.

3.5. Funcionamiento de mexcuda

La parte secuencial de una aplicación se ejecuta sobre la CPU (comúnmente denominada host) y la parte más costosa del cálculo se ejecuta sobre la GPU (que se denomina device).

Dada la naturaleza heterogénea del modelo de programación CUDA, una secuencia típica de operaciones para un programa CUDA C es:

- Declarar y asignar la memoria del host y del device.
- Inicializar los datos del host.
- Transferir datos desde el host al device.
- Ejecutar uno o más kernels (llamadas a la GPU usando CUDA).
- Transferir los resultados del device al host.

Para comprender un poco mejor su funcionamiento el mejor ejemplo es el programa “TimesTwo”, que multiplica por dos cada elemento, del array que le entra, en un hilo diferente de la GPU.

Lo primero es llamar a la función mxInitGPU en la entrada de su archivo MEX. Esto garantiza que el dispositivo GPU se inicialice correctamente y sea conocido por MATLAB.

```
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, mxArray const *prhs[])
{
/* Declaramos variables. */
int const threadsPerBlock = 256;
mxInitGPU();

/* Verificamos que A realmente es una array de double antes de
extraer el puntero. */
if (mxGPUGetClassID(A) != mxDOUBLE_CLASS) {
      mexErrMsgIdAndTxt(errId, errMsg);
}

/* Extraemos el puntero al dato de entrada en el device. */
d_A = (double const *)(mxGPUGetDataReadOnly(A));

N = (int)(mxGPUGetNumberOfElements(A));

/* Llamamos al kernel para que ejecute la funcion "TimesTwo" */
TimesTwo<<<1, threadsPerBlock>>>(d_A, d_B, N);

/* Convierte el resultado obtenido en gpuArray como la que usa
Matlab para su envío. */
plhs[0] = mxGPUCreateMxArrayOnGPU(B);

/* Los punteros de mxGPUArray son estructuras que se relacionan con
los datos que se obtiene del device. Han de ser destruidos antes de
salir de la función MEX. */
mxGPUDestroyGPUArray(A);
mxGPUDestroyGPUArray(B);
}
```

Como vemos tras inicializar la GPU, comprobamos que los datos que entran tengan el formato correcto y procedemos a extraer los punteros para después obtener los datos calculados en la GPU.

Tras escoger el número de hilos por bloque (256 en ese ejemplo), se procede a la llamada del kernel con la función “TimesTwo”, con la que enviaremos “d_A” y recibiremos “d_B” que será el array multiplicado por dos.

En el siguiente trozo de código puede verse como funciona la ejecución de un kernel y como se usan los hilos.

```
#include "mex.h"
#include "gpu/mxGPUArray.h"

/* Device code */
void __global__ TimesTwo(double const * const A,
                         double * const B,
                         int const N)
{
    /* Calculate the global linear index, assuming a 1-d grid. */
    int const i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N) {
        B[i] = 2.0 * A[i];
    }
}
```

“TimesTwo” es el programa que va a ejecutarse en la GPU por lo que hay que poner asignarle la función “__global__” que indica que ha sido invocada desde el Host (CPU).

En este caso, las variables "i" y "N" serán almacenadas por cada hilo en un registro, y los punteros "A" e "B" serán punteros asignados a la memoria del device.

Sólo hay dos líneas en nuestro kernel “TimesTwo”. Como se ha mencionado anteriormente, el kernel es ejecutado por múltiples hilos en paralelo. Si queremos que cada hilo procese un elemento de la matriz resultante, entonces necesitamos un medio para distinguir e identificar cada hilo.

CUDA define las variables "blockDim", "blockIdx" y "threadIdx".

La variable predefinida "blockDim" contiene las dimensiones de cada bloque de hilos como se especifica en el segundo parámetro de la llamada al kernel (ThreadsPerBlock).

Las variables predefinidas "threadIdx" y "blockIdx" contienen el índice del hilo dentro de su bloque de hilos y el bloque de hilos dentro del grid, respectivamente.

Antes de utilizar este índice para acceder a los elementos del array, su valor se comprueba con el número de elementos “n”, para asegurarse que no hay accesos a la memoria fuera de los límites permitidos.

“if (i **<** N**)”**

Esta comprobación es necesaria para los casos en los que el número de elementos en un array no sea divisible por el tamaño del bloque de hilos. Esto podría resultar en que el número de hilos ejecutados por el kernel fuera superior al tamaño del array.

“B**[**i**] =** 2.0 ***** A**[**i**];”**

La segunda línea del kernel realiza el trabajo de "TimesTwo", multiplicando cada elemento por dos.

El resultado sería el siguiente:

```
Command Window
>> mexcuda TimesTwo_CUDA.cu
Building with 'NVIDIA CUDA Compiler'.
MEX completed successfully.
>> x = ones(4,4,'gpuArray');
>> y = TimesTwo_CUDA(x)

y =

     2     2     2     2
     2     2     2     2
     2     2     2     2
     2     2     2     2
```

3.6. Paralelización del algoritmo LMS

Los filtros LMS se basan en la minimización del error cuadrático medio.
Estos filtros son estables y fáciles de implementar.
Desafortunadamente, la paralelización de este algoritmo, especialmente en los sistemas informáticos paralelos de memoria distribuida (Clusters), no es tan obvia.
Una desventaja principal del algoritmo LMS es la convergencia lenta de este planteamiento.

Hay varias variantes de LMS incluyendo PNLMS (Proportional Normalized Least Mean Square) que se centran en mejorar la débil convergencia del método LMS original, disminuyendo a su vez el error obtenido, el problema es que son más complejos.

El procedimiento de adaptación del filtro requiere un cálculo significativo y un coste de tiempo, que debe ser minimizado.
La convergencia más rápida del algoritmo necesita un mayor tamaño de los vectores utilizados dentro del filtro (miles de elementos).

El elemento más complejo del proceso computacional es el procedimiento de la multiplicación de matrices.
Tras realizar la paralelización obtenemos un algoritmo concurrente que funciona como el secuencial, pero mucho más rápido.

El algoritmo paralelo se ha diseñado en C con el uso de la arquitectura CUDA para ser ejecutado en GPUs nVidia.

Este método se denomina paralelización de un solo paso y consiste en utilizar la ordenación del cálculo paralelo para acelerar el elemento (operaciones) que más recursos consuma.
En el filtro LMS, este elemento representa el producto de matrices.

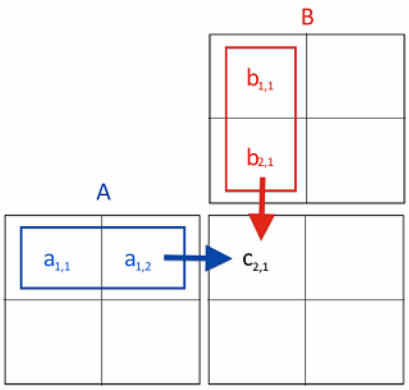


Fig. 3.8 Multiplicación matricial

Hay que calcular cada elemento en “C”, y cada uno de los otros es independiente, por lo que podremos paralelizarlo de manera eficaz.

En las pruebas experimentales que veremos en el siguiente apartado, se ha utilizado el algoritmo LMS para identificar un amplificador.

“xBB” contiene los componentes en fase y cuadratura de los valores de entrada del amplificador.
En “yBB” se almacenan los valores IQ de salida de este amplificador.

Este algoritmo nos permite conocer la ecuación que mejor relaciona la entrada y la salida del amplificador.

Como hemos visto anteriormente, $\mathrm{y(n)} = w^H(n) \cdot x(n)$, caracteriza la ecuación de salida del algoritmo respecto a los valores de entrada, los cuales multiplicamos por unos coeficientes que nos estimarán esta señal deseada.

La matriz “X” contiene los valores de “xBB” elevados al cuadrado, al cubo, etc., además de diversos términos no-lineales de este mismo array.

xBB

12288x1 complex double

	1
1	0.0143 + 0.0558i
2	0.0084 - 0.0123i
3	0.0103 - 0.0944i
4	0.0164 - 0.1869i
5	0.0220 - 0.2837i
6	0.0222 - 0.3769i

X

12288x1024 complex double

	1	2
1	0.1369 + 0.2965i	0.2081 + 0.2101i
2	0.0532 + 0.3681i	0.1369 + 0.2965i
3	-0.0271 + 0.4128i	0.0532 + 0.3681i
4	-0.0890 + 0.4226i	-0.0271 + 0.4128i
5	-0.1209 + 0.3945i	-0.0890 + 0.4226i
6	-0.1161 + 0.3305i	-0.1209 + 0.3945i

yBB

12288x1 complex double

	1
1	0.0008 + 0.0130i
2	0.0001 - 0.0042i
3	0.0026 - 0.0273i
4	0.0048 - 0.0630i
5	0.0066 - 0.1014i
6	0.0056 - 0.1343i

Por lo que, en resumen, la finalidad de la aplicación de este algoritmo, es identificar un amplificador no-lineal de radiofrecuencia, utilizando los datos en fase y cuadratura (de entrada y salida) que nos proporciona este mismo.

3.7. Implementación del LMS en CUDA

El algoritmo Least Mean Square se basa principalmente en la multiplicación, suma y resta de matrices con datos complejos.
Dependiendo del número de coeficientes que queramos usar, la complejidad de los cálculos aumenta pues las matrices han de procesarse muchas más veces.

Vamos a explicar cómo se realizan mediante la programación en CUDA C.

Empezamos con las "Device Function" que serán las funciones que se realizarán paralelamente en las GPU.
Como vemos, estas refieren a los "thread" pues cada cálculo va a realizarse en uno diferente. (Es la forma en que obtenemos mayor eficiencia de procesado).
La primera llamada a CUDA va a obtener los valores de "error" y coeficientes "w". Vamos a ver qué dos funciones ejecuta esta llamada.

- La primera es "yBB_Estimation" que va a multiplicar el vector de coeficientes y los datos de la señal entrante para obtener una primera estimación de la señal de salida.

```
__global__ void yBB_Estimation(cuDoubleComplex *yest, cuDoubleComplex *w,
cuDoubleComplex const *X, int ncoefs)
{
        __shared__ cuDoubleComplex lyest[1024];

        unsigned int tid = threadIdx.x;
        unsigned int i = threadIdx.x;
        lyest[tid].x = 0; lyest[tid].y = 0;

        if(i<ncoefs)
                lyest[tid] = cuCmul( w[tid] , X[tid]);

        __syncthreads();

        for (unsigned int s=blockDim.x/2; s>0; s>>=1)  // Reduction
        {
                if (tid < s) {
                        lyest[tid] = cuCadd(lyest[tid],  lyest[tid+s]);
                }
                __syncthreads();
        }

        if (tid == 0)
                yest[0] = lyest[0]; // yBB_est(nrow); in MATLAB
}
```

Detalle del funcionamiento de la función yBB_Estimation:

```
__shared__ cuDoubleComplex lyest [1024];
```

Se declara un Buffer de memoria compartida llamada "lyest".
Este Buffer se usará para almacenar cada suma producida en los hilos que se estén ejecutando.

```
unsigned int tid = threadIdx.x;
unsigned int i = threadIdx.x;
lyest[tid].x = 0; lyest[tid].y = 0;
```

Declaramos el tamaño del array threadsPerBlock para que cada hilo en el bloque tenga un lugar para almacenar su resultado temporal.

```
if(i<ncoefs)
```

Cuando todos los hilos terminan de realizar sus operaciones, el bloque termina. Cuando todos los bloques han terminado, el kernel termina y finaliza la llamada a la GPU.

```
lyest[tid] = cuCmul( w[tid], X[tid]);
```

"CuCmul" pertenece al header "cuComplex.h" y la función multiplica los datos complejos de la siguiente forma:

```
cuDoubleComplex prod;
prod = make_cuDoubleComplex ((cuCreal(x) * cuCreal(y)) -
                             (cuCimag(x) * cuCimag(y)),
                             (cuCreal(x) * cuCimag(y)) +
                             (cuCimag(x) * cuCreal(y)));
return prod;
```

```
__syncthreads()
```

Esta función espera hasta que todos los threads dentro del mismo bloque hayan terminado.

Si se asigna memoria compartida a un kernel, el array sólo será visible para un bloque de threads, así que cada bloque tendrá su propio bloque de memoria compartida.

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1)  // Reduction
{
        if (tid < s) {
                lyest[tid] = cuCadd(lyest[tid],  lyest[tid+s]);
        }
```

Realizamos la “reducción”, que es el proceso general de recibir un array de entrada y realizar algunos cálculos que producirán un array más pequeño con los resultados.

La idea general es que cada thread sumará dos de los valores en “lyest[]” y almacenará el resultado de nuevo en “lyest[]”.
Puesto que cada hilo combina dos entradas en una, terminamos este paso con la mitad de entradas de las que empezamos.

En el siguiente paso, hacemos lo mismo en la mitad restante.

Continuamos de esta forma en pasos de log2(threadsPerBlock) hasta que tengamos la suma de cada entrada en “lyest[]”.

Un ejemplo claro de reducción es suponer que hay ocho entradas en lyest[] y, como resultado de la reducción, se obtiene el valor 4.

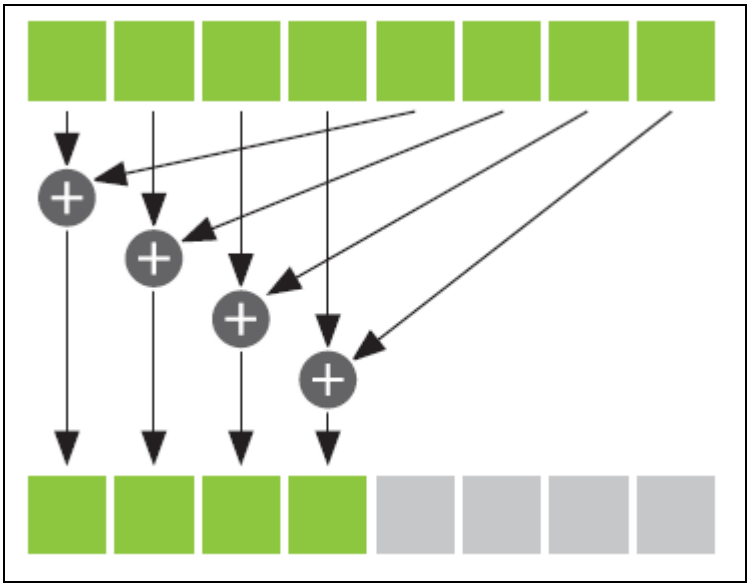


Fig. 3.9 Reducción de la suma.

```
__syncthreads()
```

Antes de poder leer los valores que acabamos de almacenar en "lyest[]", debemos asegurarnos de que cada hilo que necesite escribir en "lyest[]" ya lo haya hecho. "__syncthreads()" después de la reducción asegura que esta condición se cumple.

```
if(tid==0)
        yest[0] = lyest[0];
```

Guardamos el resultado en el thread con "tid==0" pues, como sólo hay un número que necesita ser escrito en la memoria global, no se necesitará más de un thread para esta operación.

Es posible que cada thread pudiera realizar esta escritura y el programa seguiría funcionando, pero al hacerlo crearía una cantidad innecesariamente grande de tráfico de memoria para escribir un solo valor.
Para simplificar, elegimos el thread con el índice "0", aunque se podría haber elegido cualquier "tid" para escribir "lyest[0]" en la memoria global.

- La segunda es “error_Estimation” que va a sumar los valores calculados de los coeficientes del filtro (weights) tras obtener los valores de “deltaw” y también va a calcular el error:

```
__global__ void error_Estimation(cuDoubleComplex *deltaw, cuDoubleComplex const
*yBB, cuDoubleComplex *yest ,
                                        cuDoubleComplex const *X, int ncoefs,
cuDoubleComplex *w, double mu, cuDoubleComplex *d_error)
{

        int idx = threadIdx.x;

        cuDoubleComplex error = cuCsub(*yBB, *yest);

        if(idx<ncoefs)
        {
                deltaw[idx] = cuCmul(cuConj(X[idx]) , error);
                deltaw[idx].x = deltaw[idx].x*mu;
                deltaw[idx].y = deltaw[idx].y*mu;
                w[idx] = cuCadd( w[idx] ,  deltaw[idx]);
        }

        if(idx==0)
        {
                d_error[0] = error;
        }
}
```

Detalle del funcionamiento de la función “error_Estimation”:

```
cuDoubleComplex error = cuCsub(*yBB, *yest);
```

"cuCsub" pertenece al header "cuComplex.h" y la función resta los datos complejos de la siguiente forma:

```
return make_cuDoubleComplex (cuCreal(x) - cuCreal(y),
                             cuCimag(x) - cuCimag(y));
```

Con esa resta obtenemos el error, véase la diferencia entre la señal deseada y la señal de salida estimada.

```
deltaw[idx] = cuCmul(cuConj(X[idx]) , error);
deltaw[idx].x = deltaw[idx].x*mu;
deltaw[idx].y = deltaw[idx].y*mu;
w[idx] = cuCadd( w[idx] ,  deltaw[idx]);
```

En esta parte del código, multiplicamos el conjugado de la señal de entrada “X” por el error obtenido anteriormente, de la misma forma que se haría en Matlab:

```
deltaw=uH'*error(nrow);
```

Finalmente se suman los valores de “w” y de delta (anteriormente multiplicada por el factor de paso “mu”).

```
w=w+deltaw*mu;
```

La segunda llamada a CUDA desde Matlab va a multiplicar los valores de la señal “X” por los coeficientes “w” recién calculados.
Esta llamada a CUDA va a tardar muchísimo menos pues solo realizará el producto de dos arrays:

```
__global__ void yBB_Estimation_Post(cuDoubleComplex *yest_post, cuDoubleComplex
const *X, cuDoubleComplex const *W, int ncoefs)
{
	__shared__ cuDoubleComplex lyest_post[256];
	unsigned int tid = threadIdx.x;
	unsigned int stid = blockIdx.x*ncoefs + tid;

	lyest_post[tid].x = 0; lyest_post[tid].y = 0;

	if(tid<ncoefs)
		lyest_post[tid] = cuCmul( X[stid] , W[tid]);

	__syncthreads();

	for (unsigned int s=blockDim.x/2; s>0; s>>=1)   // Reduction
	{
		if (tid < s) {
			lyest_post[tid] = cuCadd(lyest_post[tid],  lyest_post[tid+s]);
		}
		__syncthreads();
	}

	if (tid == 0)
		yest_post[blockIdx.x] = lyest_post[0];

}
```

Detalle del funcionamiento de la función “yBB_Estimation_Post”:

El funcionamiento es casi idéntico al explicado anteriormente pues es solamente una multiplicación de arrays, la diferencia reside a la hora de almacenar los datos en “yest_post”.

```
if (tid == 0)
	yest_post[blockIdx.x] = lyest_post[0];
```

En este caso en vez de utilizar solamente el threadIdx, estamos utilizando también el “blockIdx” pues a diferencia de antes, lo que obtenemos del cálculo es un vector de resultados y no un solo valor. En la anterior función cada vez se escogía un valor del vector, en esta queremos escogerlos todos pues buscamos el resultado final de la multiplicación, no un coeficiente.

3.8. Resultados obtenidos

Ejemplo de Algoritmo LMS en MATLAB (CPU):

Este ejemplo de Least Mean Square Algorithm es propiedad del tutor del proyecto. La función “gml_LS_Mgeneration” devuelve una matriz compleja de 12288x210 elementos que sirven para calcular los coeficientes necesarios para este filtro adaptativo. Hemos escogido 210 coeficientes para este proyecto.

```
clear all; clc; close all;
rootpath=cd;
addpath(rootpath);
addpath([rootpath '\toolbox']);

load('PAcreeLTE20M.mat');

Ncoef_poly=10;
delays_poly=[-10:1:10];
X=gml_LS_Mgeneration('MEMPOLY2',xBB,Ncoef_poly,delays_poly,0,0);

[Nrows,Ncols]=size(X)

%%%%% LS solved with LMS in .m
mu=0.015;

disp(' '); disp('LS: Doing LMS');

w=complex(zeros(Ncols,1));
yBB_est=zeros(Nrows,1);
error=zeros(Nrows,1);

Tstart=tic;
for nrow=1:Nrows
   %%% call-1 CUDA
   uH=X(nrow,:);
   yBB_est(nrow)=uH*w;
   error(nrow)=yBB(nrow)-yBB_est(nrow);
   deltaw=uH'*error(nrow);
   w=w+deltaw*mu;
   %%%% return from CUDA
end
T_LMS=toc(Tstart);
fprintf('->time1= %f\n',T_LMS);

Tstart=tic;
   %%% call-2 CUDA
yBB_est_post=X*w; % estimation using the last coefficients
   %%% return from CUDA
T_post=toc(Tstart);
fprintf('->time2= %f\n',T_post);

fprintf('->NMSE estimation post= %f\n',dpd_Qmeasurements(yBB,yBB_est_post,'NMSE'));

plot(abs(xBB),abs(yBB),'.b')
hold on
plot(abs(xBB),abs(yBB_est_post),'.r')
```

Resultado del LMS en MATLAB (CPU):

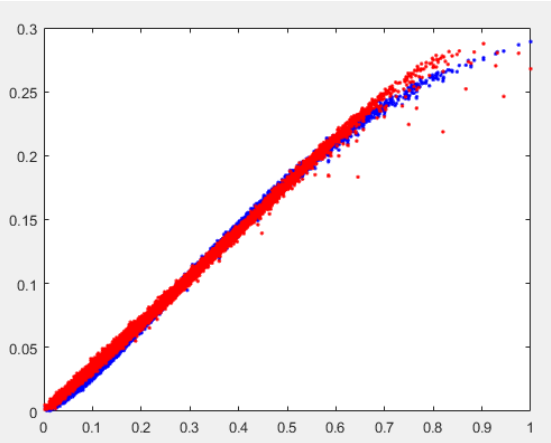
```
Nrows =

        12288


Ncols =

    210


LS: Doing LMS
->time1= 0.290911
->time2= 0.006496
->NMSE estimation post= -28.625438
```



--- plot(abs(xBB),abs(yBB),'.b')
--- plot(abs(xBB),abs(yBB_est_post),'.r')

El valor de NSME (normalized mean square error) encontrado es una estimación de las desviaciones totales entre los valores que esperábamos y los medidos.

$$NMSE_{log} = 10 * \log_{10}\left(\mathrm{E}\left(\frac{(y-x)^2}{x^2}\right)\right)$$

Cuanto menor sea el valor de NSME, mayor será el rendimiento de este modelo que estamos utilizando.
Haciendo pruebas en MATLAB la estimación encontrada es de -28.62438 dB.
El tiempo de ejecución del algoritmo es de 0.290911 segundos.

He realizado el mismo algoritmo, pero utilizando la GPU y programándola con CUDA C para ver la mejoría de rendimiento.

Mi compañero Rahul ha programado este mismo algoritmo para su TFG, pero en lenguaje .C y utilizando la CPU, por ello le estoy agradecido pues ha permitido que lo compare con mis resultados.

Las tres adaptaciones del algoritmo LMS obtienen el mismo resultado de NSME por lo que puede constatarse que funcionan correctamente y que los valores obtenidos son idénticos.

El siguiente paso es probarlos y determinar qué versión es la más óptima para grandes cantidades de datos y coeficientes.

El algoritmo realiza dos llamadas a CUDA, en la primera se obtienen los valores de los coeficientes (pesos) que van a adaptar la señal y en la segunda se multiplican estos valores por la señal de entrada y se obtiene la estimación final de la señal que deseamos. En las siguientes mediciones observaremos la duración de estos dos procesos.

Tras varias pruebas utilizando **210 coeficientes**, se ha obtenido los siguientes resultados:

Tabla 3.1. Tiempos de ejecución del algoritmo en distintos tipos de procesador

Tiempos de ejecución (segundos)	Versión Matlab (CPU)	Versión CUDA (GPU)	Versión .C (CPU)
PC Laboratorio 225 (GTX 660)	0.443244 (1ª) 0.007568 (2ª)	0.165281 (1ª) 0.003169 (2ª)	0.091177 (total)
Ordenador Portátil (GeForce 930M)	0.290911 (1ª) 0.006496 (2ª)	0.146715 (1ª) 0.002908 (2ª)	0.064149 (total)
PC sobremesa personal (GTX 970)	0.225325 (1ª) 0.004054 (2ª)	0.124273 (1ª) 0.002026 (2ª)	0.055408 (total)

La primera y segunda llamada a CUDA corresponden a: (1ª) y (2ª).
En .C todo el código se ejecuta en una sola llamada: (total), por lo que el tiempo de ejecución será equivalente a la suma de (1ª) y (2ª).

Podemos observar que los resultados obtenidos son gratamente positivos pues el algoritmo se está ejecutando, como mínimo, el doble de rápido.

Hay que matizar varios detalles respecto a los resultados:

- Donde mayor diferencia de tiempo se observa es en el caso del PC Laboratorio 225, en el que la versión de CUDA se ejecuta casi tres veces más rápido que Matlab.
  Esto puede darse por la configuración hardware que posee este ordenador pues la tarjeta gráfica “GTX 660” pese a tener unos cuantos años (2012) es una tarjeta de gama media-alta por lo que su rendimiento es muy elevado.
  En comparación con el procesador “Intel i7 920” (2008) que es de gama media, la capacidad de computación es bastante inferior ya que en cuatro años la tecnología ha evolucionado mucho.

- Utilizando componentes informáticos de similar categoría y años vemos que el PC de sobremesa personal obtiene los mejores resultados en cuanto a velocidad de ejecución.
  Esta tarjeta gráfica “GTX 970” posee 1664 núcleos CUDA, óptimos para el paralelismo, que aumentan por dos el rendimiento que obtendría el procesador “i5-6500”.

- El ordenador portátil ofrece también buenos resultados puesto que posee componentes actuales y de similar categoría.

- El tiempo de ejecución del programa en .C es muy superior a las pruebas realizadas en CUDA. Este sorprendente resultado se explica porque estamos usando pocos coeficientes y el “Core Speed” o frecuencia básica del CPU (3.20 GHz), es mucho más potente que la de la GPU (1.05 GHz).

- Una posible solución es montar ordenadores dedicados exclusivamente a la computación GPU pues las tarjetas gráficas estarían libres de carga (no habría ningún dato precompilado ni se harían cargo de hacer funcionar el monitor).

Este algoritmo está trabajando con matrices de 12288x210 valores complejos.

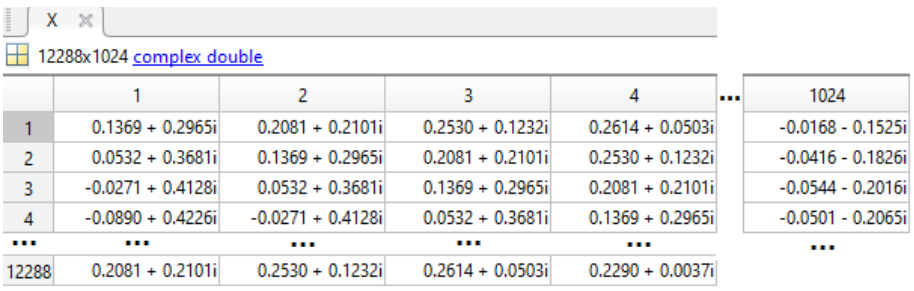
La paralelización ha tenido éxito, pero si queremos observar mejores resultados y, por lo tanto, mayor aumento de velocidad, tenemos que trabajar con más coeficientes ya que en la CPU estos cálculos se van a realizar secuencialmente y a mayor número de elementos, más tiempo de procesado. En la GPU en cambio al trabajar con tantos hilos en paralelo, estos cálculos se realizan al mismo tiempo en grandes bloques.

Tras probar este algoritmo en CUDA utilizando cantidades distintas de coeficientes observamos lo siguiente:

Tabla 3.2. Tiempos de ejecución del algoritmo con distintos coeficientes

Nº Coeficientes	Duración con MATLAB (s)	Duración con CUDA (s)	Speedup
21	0,139905	0,139693	1,002
64	0,161046	0,129507	1,244
128	0,178201	0,135671	1,313
256	0,226118	0,149717	1,510
512	0,364946	0,168932	2,160
1024	0,715634	0,223281	3,205

Cuando hablamos de que se está utilizando 21 o 1024 coeficientes, lo que CUDA está procesando son matrices de 12288xcoeficientes por lo que, en el último ejemplo de 1024 coeficientes, la matriz procesada sería esta:


X

12288x1024 complex double

	1	2	3	4	...	1024
1	0.1369 + 0.2965i	0.2081 + 0.2101i	0.2530 + 0.1232i	0.2614 + 0.0503i		-0.0168 - 0.1525i
2	0.0532 + 0.3681i	0.1369 + 0.2965i	0.2081 + 0.2101i	0.2530 + 0.1232i		-0.0416 - 0.1826i
3	-0.0271 + 0.4128i	0.0532 + 0.3681i	0.1369 + 0.2965i	0.2081 + 0.2101i		-0.0544 - 0.2016i
4	-0.0890 + 0.4226i	-0.0271 + 0.4128i	0.0532 + 0.3681i	0.1369 + 0.2965i		-0.0501 - 0.2065i
...	...	...	...	...		...
12288	0.2081 + 0.2101i	0.2530 + 0.1232i	0.2614 + 0.0503i	0.2290 + 0.0037i		

Fig. 3.10 Matriz procesada por la GPU

El siguiente gráfico nos muestra la aceleración de CUDA con respecto al uso de la CPU con MATLAB.

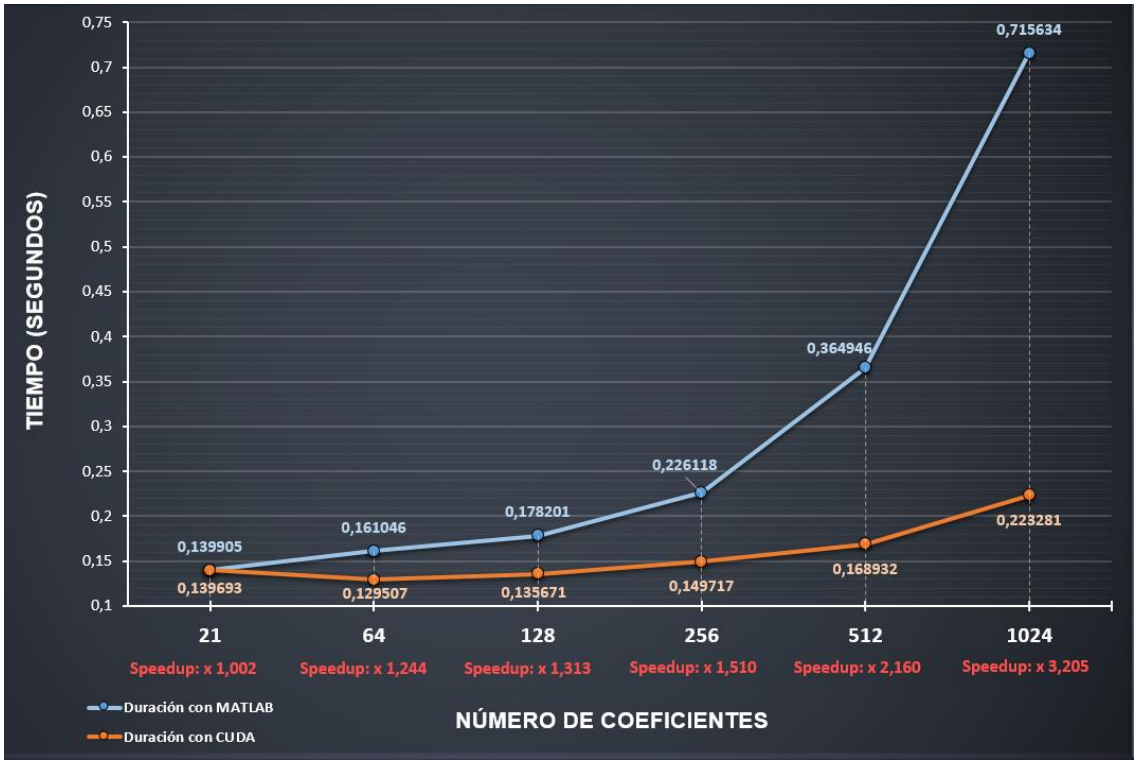


Fig. 3.11 Mejora de CUDA respecto a la CPU

Comprobamos que a mayor número de coeficientes, vease mayor tamaño de matrices, el tiempo de ejecución en MATLAB aumenta considerablemente hasta casi seis veces entre los 21 coeficientes y los 1024.

En cambio gracias a CUDA el tiempo de procesado apenas varía. El único problema de cuda es que con matrices pequeñas y poca cantidad de datos a procesar, este apenas ofrece mejoría pues apenas se usa la paralelización.

Vemos que con 21 coeficientes (matrices de 12288x210), la CPU es suficientemente potente como para igualar el rendimiento de la GPU.

De otro lado, con 1024 coeficientes, la CPU triplica el tiempo de procesado respecto a la GPU con lo que es realmente útil el uso de esta última.

Ahora volviendo al código en .C que me ha prestado Rahul, si hacemos pruebas pero en este caso con **1024 coeficientes**, el algoritmo en CUDA se vuelve más eficaz que en .C.

Tabla 3.3. Tiempos de ejecución del algoritmo con 1024 coeficientes

Tiempos de ejecución (segundos)	Versión Matlab (CPU)	Versión CUDA (GPU)	Versión .C (CPU)
PC Laboratorio 225 (GTX 660)	1.248812	0.235045	0.318960
Ordenador Portátil (GeForce 930M)	0.826053	0.230159	0.284348
PC sobremesa personal (GTX 970)	0.715634	0.223281	0.260961

Puede observarse que la aceleración de CUDA en el PC del Laboratorio es de 5 veces respecto a Matlab y está por encima de .C.

Como se ha comentado antes, las mejorías en el PC de sobremesa y el portatil son algo menores pues tanto la CPU como la GPU son muy potentes y no se aprecia tanto la diferencia.

Algo a tener en cuenta con el algoritmo LMS es que para calcular la estimación de la señal de salida y los coeficientes, es necesario calcular los valores utilizando los anteriores y así tantas veces como datos de entrada tengamos.

Esto puede penalizar levemente al paralelismo de CUDA ya que la manera óptima de utilización se obtiene realizando muchas iteraciones en un bucles en los que los datos no dependen de su anterior iteración.

También hay que tener en cuenta que se ha trabajado con un fichero de 12288 datos complejos por lo que utilizando mayor cantidad de datos y algún método de optimización del código en CUDA, los resultados y la aceleración serían mucho superiores a los encontrados.

Conclusiones

Este proyecto ha servido para ver el potencial de la arquitectura CUDA y entender su funcionamiento aplicándola al algoritmo de un filtro adaptativo.

El lenguaje CUDA C puede aplicarse en cualquier sector como se ha explicado en capítulos anteriores, cualquier código que tenga elementos paralelizables es perfecto para su conversión.

El verdadero desafío con la optimización de los algoritmos es determinar qué partes se pueden realizar en la GPU, y qué partes se deben realizar en la CPU.

El algoritmo LMS tiene el propósito de reducir el ruido presente en la señal de entrada de tal modo que la señal de salida del filtro se aproxime lo más posible a una señal deseada. Esto lo convierte en un buen ejemplo para aplicar la paralelización pues necesita mayor número de iteraciones para llegar a la convergencia y a su vez mayor número de coeficientes.

Este algoritmo ha sido totalmente adaptado a CUDA y se ha demostrado que, a mayor exigencia computacional, mejor rinde la GPU frente a la CPU.

Para obtener mejores resultados podría utilizarse un servidor GPU dedicado, pues se dispondría de la mejor tecnología actual. En nuestros ejemplos la tarjeta gráfica está trabajando también con el monitor, por lo que no está totalmente libre de trabajo.

Me ha resultado muy interesante aprender desde cero el funcionamiento de esta arquitectura, pues hasta el momento desconocía totalmente sus ventajas y veo que puedo aplicarla a lo que quiera, desde el procesado de señales que hemos estudiado a lo largo de la carrera, como al sector financiero o de defensa.

La computación paralela está cada vez más en auge, las principales empresas la utilizan y como se ha mencionado antes, Nvidia ha creado la red neuronal artificial más grande del mundo mediante el uso de múltiples GPUs.

Tengo claro que van a aparecer tecnologías que desbanquen a esta, pero por el momento CUDA es el futuro y voy a seguir adquiriendo sus conocimientos.

Bibliografía

[1] Nvidia, *CUDA C programming guide*. PG-02829-001_v8.0, January 2017.

[2] Jason Laska, *Writing C Functions in MATLAB (MEX-Files).*

[3] KV, NSmith, *Accelerating MATLAB with CUDA™ Using MEX Files, WP-*03495-001_v01, September 2007.

[4] Joss Knight, *Calling CUDA-accelerated Libraries from MATLAb: A Computer Vision Example*, July 29, 2014.

[5] Nvidia, *Accelerate your Application with CUDA C.*

[6] Kirk, D., Hwu, *Programming Massively Parallel Processors: A Hands-on Approach.* 1 edn. Morgan Kaufmann (February 2010)

[7] Wojciech Bożejko, Andrzej Dobrucki, Maciej Walczyński, *Parallelizing of digital signal processing with using GPU,* 25 Sept. 2010.

[8] Nikolaos Ploskas Nikolaos Samaras*, GPU Programming in MATLAB*, chapter 8, “MATLAB MEX functions containing CUDA code”, pp. 219-239, edn. Morgan Kaufmann, 28th July 2016.

[9] Pascal Getreuer, *Writing MATLAB C/MEX Code*, April 2010.

[10] http://www.nvidia.es/object/gpu-computing-applications-es.html, *Aplicaciones aceleradas en la GPU*.

[11] http://www.nvidia.es/object/cuda-parallel-computing-es.html, *Procesamiento paralelo CUDA.*

[12] Nvidia, *CUBLAS LIBRARY*, DU-06702-001_v8.0 | January 2017.

[13] Jagdamb Behari Srivastava, R.K. pandey, Jitendra Jain, *Implementation of Digital Signal Processing Algorithm in General Purpose Graphics Processing Unit (GPGPU),* Vol. 1, Issue 4, June 2013.

[14] Maciej WALCZYŃSKI, Wojciech BOŻEJKO, *Noise reduction with using parallel algorithms,* Zamek Książ - Wałbrzych, Poland, 6-9 June 2010.

[15] Egil Fykse, *Performance Comparison of GPU, DSP and FPGA implementations of image processing and computer vision algorithms in embedded systems,* June 2013.

[16] Manuel Alejandro Ayala, Nadezhda Rocío Córdova, *Diseño e implementación de un filtro con algoritmo adaptativo en fpga para la cancelación de ruido,* año 2016.

[17] Jorge Lorente, Miguel Ferrer, *Real-time adaptive algorithms using a Graphics Processing Unit*

[18] B. Widrow, S. D. Stearns, *Adaptive Signal Processing*, Prentice-Hall Signal Processing Series, 1985.

[19] A. Gonzalez, C. Gonzalez, F. J. MartinezZaldivar, C. Ramiro, S. Roger, A. M. Vidal, *The impact of GPU/Multicore in Signal Processing: a quantitative approach*, in Waves, vol. 3, pp. 96-106, 2011.

Anexos

Ejemplo TimesTwo en CUDA C:

```
/* Example of how to use the mxGPUArray API in a MEX file.  This example shows how to write a
MEX function that takes a gpuArray input and returns a gpuArray output, e.g. B=mexFunction(A).
 * Copyright 2012 The MathWorks, Inc. */

#include "mex.h"
#include "gpu/mxGPUArray.h"

/* Device code */
void __global__ TimesTwo(double const * const A,
                         double * const B,
                         int const N)
{
    /* Calculate the global linear index, assuming a 1-d grid. */
    int const i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N) {
        B[i] = 2.0 * A[i];
    }
}

/* Host code */
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, mxArray const *prhs[])
{
    /* Declare all variables.*/
    mxGPUArray const *A;
    mxGPUArray *B;
    double const *d_A;
    double *d_B;
    int N;
    char const * const errId = "parallel:gpu:mexGPUExample:InvalidInput";
    char const * const errMsg = "Invalid input to MEX file.";

    /* Choose a reasonably sized number of threads for the block. */
    int const threadsPerBlock = 256;
    int blocksPerGrid;

    /* Initialize the MathWorks GPU API. */
    mxInitGPU();

    /* Throw an error if the input is not a GPU array. */
    if ((nrhs!=1) || !(mxIsGPUArray(prhs[0]))) {
        mexErrMsgIdAndTxt(errId, errMsg);
    }

    A = mxGPUCreateFromMxArray(prhs[0]);

    /* Verify that A really is a double array before extracting the pointer. */
    if (mxGPUGetClassID(A) != mxDOUBLE_CLASS) {
        mexErrMsgIdAndTxt(errId, errMsg);
    }

    /*
     * Now that we have verified the data type, extract a pointer to the input
     * data on the device. */
    d_A = (double const *)(mxGPUGetDataReadOnly(A));
```

```
    /* Create a GPUArray to hold the result and get its underlying pointer. */
    B = mxGPUCreateGPUArray(mxGPUGetNumberOfDimensions(A),
                            mxGPUGetDimensions(A),
                            mxGPUGetClassID(A),
                            mxGPUGetComplexity(A),
                            MX_GPU_DO_NOT_INITIALIZE);
    d_B = (double *)(mxGPUGetData(B));

    /* Call the kernel using the CUDA runtime API. We are using a 1-d grid here, and it would be
possible for the number of elements to be too large for the grid. For this example we are not guarding
against this possibility. */
    N = (int)(mxGPUGetNumberOfElements(A));
    blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    TimesTwo<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, N);

    /* Wrap the result up as a MATLAB gpuArray for return. */
    plhs[0] = mxGPUCreateMxArrayOnGPU(B);

    /* The mxGPUArray pointers are host-side structures that refer to device data. These must be
destroyed before leaving the MEX function. */
    mxGPUDestroyGPUArray(A);
    mxGPUDestroyGPUArray(B);
}
```

Resultado de TimesTwo en CUDA C:

Command Window

```
>> mexcuda TimesTwo_CUDA.cu
Building with 'NVIDIA CUDA Compiler'.
MEX completed successfully.
>> x = ones(4,4,'gpuArray');
>> y = TimesTwo_CUDA(x)

y =

     2     2     2     2
     2     2     2     2
     2     2     2     2
     2     2     2     2
```

Pruebas de CUDA utilizando la Ecuación Normal:

Con tal de obtener más resultados y poder contrastarlos con la base del proyecto, que es la paralelización con CUDA del algoritmo adaptativo LMS, he creado algunos ejemplos de ecuación normal en .C, CUDA C y MATLAB.

Dada una ecuación de matrices $\boldsymbol{Ax} = \boldsymbol{b}$, $\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$

La ecuación normal es aquella que minimiza la suma de las diferencias cuadradas entre los lados izquierdo y derecho de la igualdad y se denota de la siguiente manera: $\boldsymbol{A^T A x} = \boldsymbol{A^T b}$.
En este caso $\boldsymbol{A^T A}$ es una matriz normal lo que significa que $\boldsymbol{A^* A} = \boldsymbol{A A^*}$.
($\boldsymbol{A^*}$ es la matriz traspuesta conjugada).

Programa en C:

```
#if !defined(_WIN32)
#define dgemm dgemm_
#endif

#include "mex.h"
#include "blas.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
*prhs[])
{
    double *A, *B, *C; /* pointers to input & output matrices*/
    size_t m,n,p;      /* matrix dimensions */
    char *chn = "N";
    double one = 1.0, zero = 0.0;

    A = mxGetPr(prhs[0]); /* first input matrix */
    B = mxGetPr(prhs[1]); /* second input matrix */
    /* dimensions of input matrices */
    m = mxGetM(prhs[0]);
    p = mxGetN(prhs[0]);
    n = mxGetN(prhs[1]);

    if (p != mxGetM(prhs[1])) {
        mexErrMsgIdAndTxt("MATLAB:matrixMultiply:matchdims",
                "Inner dimensions of matrix multiply do not
match.");
    }
    /* create output matrix C */
    plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
    C = mxGetPr(plhs[0]);
    /* Pass arguments by reference */
    dgemm(chn, chn, &m, &n, &p, &one, A, &m, B, &p, &zero, C, &m);
}
```

Este programa utiliza la librería BLAS (Basic Linear Algebra) y ejecuta “dgemm” que realiza la multiplicación de matrices de manera más eficaz que programándola a mano. Se ejecuta así: “%mex -v cblasCPU.c –lmwblas”

Matlab llamando a CUDA:

```
axBB=abs(xBB);
ayBB=abs(yBB);

M=[ones(PARAM.L.LBB,1) , axBB];
[Nrows,Ncolumns]=size(M);

mexcuda CUBLASGPU.cu -lcublas

axBBB = ones(Nrows,2');
axBBB(:,2) = axBB;
axBBT = axBBB';

tic;
res1 = CUBLASGPU(axBBT,axBBB);
res2 = CUBLASGPU(axBBT,ayBB);

toc
w = (inv(res1))*res2;
ayBB_est = axBBB*w;      % estimate result

error=zeros(Nrows,1);
for niter=1:Nrows
    error(niter)=ayBB(niter)-ayBB_est(niter);
end
```

CUDA:

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]){

mxInitGPU();

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

double *deviceA, *deviceB, *deviceC;
const double *A, *B;
double *C;
int numARows, numACols;
int numBRows, numBCols;
int numCRows, numCCols;

// define error messages //
char const * const errId = "parallel:gpu:mexMatrixMultiplication:InvalidInput";
char const * const errMsg = "Invalid input to MEX file.";

 // check input data //
if(nrhs != 2){
  mexErrMsgIdAndTxt(errId, errMsg);
}
// get input arrays //
A = (double *)mxGetData(prhs[0]);
B = (double *)mxGetData(prhs[1]);
```

```
 // find arrays dimensions //
 numARows = (int)mxGetM(prhs[0]);
 numACols = (int)mxGetN(prhs[0]);
 numBRows = (int)mxGetM(prhs[1]);
 numBCols = (int)mxGetN(prhs[1]);
 numCRows = numARows;
 numCCols = numBCols;

// initialize output array //
plhs[0] = mxCreateNumericMatrix(numCRows, numCCols, mxDOUBLE_CLASS, mxREAL); // 2-D
numeric matrix //
C = (double *)mxGetData(plhs[0]); //Pointer to real numeric data elements in array //

// allocate memory on the GPU //
cudaMalloc(&deviceA, sizeof(double) * numARows * numACols);
cudaMalloc(&deviceB, sizeof(double) * numBRows * numBCols);
cudaMalloc(&deviceC, sizeof(double) * numCRows * numCCols);

// Set arrays and perform the matrix multiplication using CUBLAS //

cublasHandle_t handle; // pointer type to an opaque structure holding the cuBLAS library context. //

//The application must initialize the handle to the cuBLAS library context by calling cublasCreate() //
cublasCreate(&handle);

//cublasSetMatrix copies a tile of rows x cols elements from matrix "A" in host memory space to matrix
"deviceA" in GPU memory space. //
cublasSetMatrix(numARows, numACols, sizeof(double), A, numARows, deviceA, numARows);
cublasSetMatrix(numBRows, numBCols, sizeof(double), B, numBRows, deviceB, numBRows);

double alpha = 1.0; //<type> scalar used for multiplication.
double beta = 0.0; // <type> scalar used for multiplication. If beta==0,

cublasDgemm(handle,
    CUBLAS_OP_N, CUBLAS_OP_N, // CUBLAS_OP_N --> the non-transpose operation is selected
    numARows, numBCols, numACols,
    &alpha,
    deviceA, numARows,
    deviceB, numBRows,
    &beta,
    deviceC, numCRows);

 // Recupero resultado // This function copies a tile of rows x cols elements from a matrix A in GPU
memory space to a matrix B in host memory space.
 cublasGetMatrix(numCRows, numCCols, sizeof(double), deviceC, numCRows, C, numCRows);

cudaEventRecord(stop);
// The function cudaEventSynchronize() blocks CPU execution until the specified event is recorded. //
cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);

printf("Elapsed time for CUDA operation: %f milliseconds\n", milliseconds);

 //Once the application finishes using the library, it must call the function cublasDestory() to release
the resources associated with the cuBLAS library context. //
 cublasDestroy(handle);
 cudaFree(deviceA);
 cudaFree(deviceB);
 cudaFree(deviceC);
```

En este programa se utiliza la librería CUBLAS (CUDA BLAS), que contiene funciones muy optimizadas para la multiplicación de matrices. En este caso

utilizaremos la función “cublasDgemm” que funciona igual que la versión en C. Pero exprimiendo al máximo el concepto de paralelización.

El programa ejecutado en .C tiene una duración de 0.011863 segundos.

El programa ejecutado en CUDA tiene una duración de 0.006530 segundos.

Observamos que al ejecutar el código utilizando una librería muy optimizada para multiplicar matrices, la aceleración del código es de más de 5 veces.

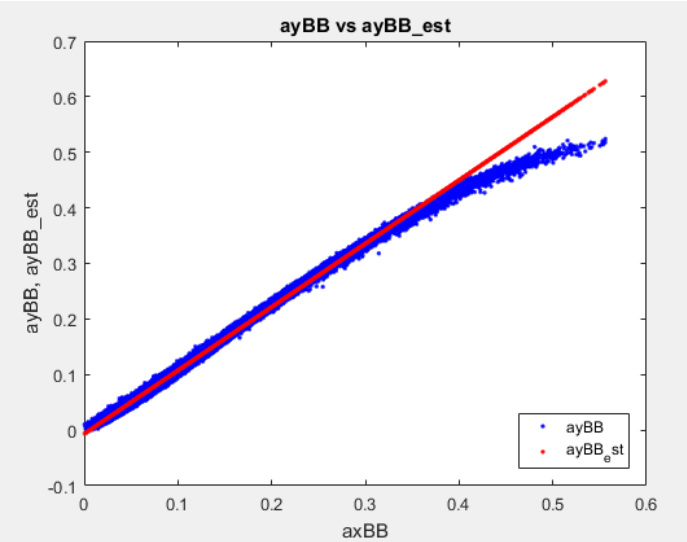


Fig. Anexos.1 Señal de salida y señal estimada

Algoritmo LMS usando CUDA:

Matlab llamando a CUDA, “GPU_MAIN_LMS.m”:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all; clc; close all;
rootpath=cd;
addpath(rootpath);
addpath([rootpath '\toolbox']);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
load('PAcreeLTE20M.mat');

Ncoef_poly=10;
delays_poly=[-10:1:10];
X=gml_LS_Mgeneration('MEMPOLY2',xBB,Ncoef_poly,delays_poly,0,0);

[Nrows,Ncols]=size(X)

% mexcuda cuda_lms_complex.cu
% mexcuda cuda_multiply.cu

%%%%% LS solved with LMS in .m
mu=0.015;

disp(' '); disp('LS: Doing LMS');

yBB_est=complex(zeros(Nrows,1));
error=zeros(Nrows,1);

r_X = reshape(X.',[],1); %Reshape de la matriz 12288x210 a una 2580480x1 para pasar los datos a
CUDA

Tstart=tic;

g_X = gpuArray(r_X);
g_yBB = gpuArray(yBB);

%%% call-1 CUDA
[w ,error] = cuda_lms_complex(g_X,g_yBB);

T_LMS=toc(Tstart);
fprintf('->time1= %f\n',T_LMS);

Tstart=tic;

g_w = gpuArray(w);% upload them to GPU

%%% call-2 CUDA
yBB_est_post=cuda_multiply(g_X,g_w); %estimation using the last coefficients

%%% return from CUDA
T_post=toc(Tstart);

fprintf('->time2= %f\n',T_post);

fprintf('->NMSE estimation post= %f\n',dpd_Qmeasurements(yBB,yBB_est_post,'NMSE'));

plot(abs(xBB),abs(yBB),'.b')
hold on
plot(abs(xBB),abs(yBB_est_post),'.r')
```

Primera llamada a CUDA, “cuda_lms_complex.cu”:

```
#include "mex.h"
#include "gpu/mxGPUArray.h"
#include <cuComplex.h>
/* Device code */
__global__ void yBB_Estimation(cuDoubleComplex *yest, cuDoubleComplex *w, cuDoubleComplex const *X, int
ncoefs)
{
	__shared__ cuDoubleComplex lyest[256];

	unsigned int tid = threadIdx.x;
	unsigned int i = threadIdx.x;
	lyest[tid].x = 0; lyest[tid].y = 0;

	if(i<ncoefs)
		lyest[tid] = cuCmul( w[tid] , X[tid]);

	__syncthreads();

	for (unsigned int s=blockDim.x/2; s>0; s>>=1)  // Reduction
	{
		if (tid < s) {
			lyest[tid] = cuCadd(lyest[tid],  lyest[tid+s]);
		}
		__syncthreads();
	}

	if (tid == 0)
		yest[0] = lyest[0]; // yBB_est(nrow); in MATLAB
}

__global__ void error_Estimation(cuDoubleComplex *deltaw, cuDoubleComplex const *yBB, cuDoubleComplex
*yest , cuDoubleComplex const *X, int ncoefs, cuDoubleComplex *w, double mu, cuDoubleComplex *d_error)
{
	int idx = threadIdx.x;
	cuDoubleComplex error = cuCsub(*yBB, *yest);  // error(nrow)=yBB(nrow)-yBB_est(nrow); in MATLAB

	if(idx<ncoefs)
	{
		deltaw[idx] = cuCmul(cuConj(X[idx]) , error);		// deltaw=X'*error(nrow); in MATLAB
		deltaw[idx].x = deltaw[idx].x*mu;
		deltaw[idx].y = deltaw[idx].y*mu;
		w[idx] = cuCadd( w[idx] ,  deltaw[idx]);		// w=w+deltaw*mu; in MATLAB
	}

	if(idx==0)
	{
		d_error[0] = error;
	}
}

/* Host code */
void mexFunction(int nlhs, mxArray *plhs[],
		 int nrhs, mxArray const *prhs[])
{
   /* Declare all variables.*/
   mxGPUArray const *X, *yBB;
   cuDoubleComplex const *d_X, *d_yBB;
   cuDoubleComplex *d_error, *d_yBB_est, *deltaw, *d_w;
   cuDoubleComplex *w_error, *w;
   int Nrows, ncoefs;
   char const * const errId = "parallel:gpu:mexGPUExample:InvalidInput";
   char const * const errMsg = "Invalid input to MEX file.";

 /* Initialize the MathWorks GPU API. */
   mxInitGPU();

   /* Throw an error if the input is not a GPU array. */
   if ((nrhs!=2) || !(mxIsGPUArray(prhs[0])))
   {
      mexErrMsgIdAndTxt(errId, errMsg);
   }
```

```
    X = mxGPUCreateFromMxArray(prhs[0]);
    yBB = mxGPUCreateFromMxArray(prhs[1]);

    /* Verify that A really is a double array before extracting the pointer. */
    if (mxGPUGetClassID(X) != mxDOUBLE_CLASS)
    {
        mexErrMsgIdAndTxt(errId, errMsg);
    }
    /* Now that we have verified the data type, extract a pointer to the input data on the device. */
    d_X = (cuDoubleComplex const *)(mxGPUGetDataReadOnly(X));
    d_yBB = (cuDoubleComplex const *)(mxGPUGetDataReadOnly(yBB));

     Nrows = (int)(mxGPUGetNumberOfElements(yBB));
     ncoefs = (int)(mxGPUGetNumberOfElements(X))/Nrows;
    /* Create a GPUArray to hold the result and get its underlying pointer. */
    w_error = (cuDoubleComplex*) malloc(sizeof(cuDoubleComplex) * Nrows);
    w = (cuDoubleComplex*) malloc(sizeof(cuDoubleComplex) * ncoefs);
    cudaMalloc((void**)&d_w, sizeof(cuDoubleComplex) * ncoefs);
    cudaMalloc((void**)&deltaw, sizeof(cuDoubleComplex) * ncoefs);
    cudaMalloc((void**)&d_error, sizeof(cuDoubleComplex) * Nrows);
    cudaMalloc((void**)&d_yBB_est, sizeof(cuDoubleComplex));
    double *w_r, *w_i;
    double *error_r, *error_i;

    plhs[0]=mxCreateDoubleMatrix(ncoefs, 1, mxCOMPLEX);
    w_r = mxGetPr(plhs[0]);
    w_i = mxGetPi(plhs[0]);
    plhs[1]=mxCreateDoubleMatrix(Nrows, 1, mxCOMPLEX);
    error_r = mxGetPr(plhs[1]);
    error_i = mxGetPi(plhs[1]);

    double mu = 0.015;
    for(int i=0; i<ncoefs; i++)
    {
            w[i].x = 0;
            w[i].y = 0;
    }

    cudaMemcpy(d_w, w, sizeof(cuDoubleComplex) * ncoefs, cudaMemcpyHostToDevice);

            cudaEvent_t start, stop;
            cudaEventCreate(&start);
            cudaEventCreate(&stop);
            cudaEventRecord(start);

            for(int i=0; i<Nrows; i++)
            {
                    yBB_Estimation<<<1,256>>>(d_yBB_est, d_w, &d_X[i*ncoefs], ncoefs);
                    error_Estimation<<<1,256>>>(deltaw, &d_yBB[i] , d_yBB_est,  &d_X[i*ncoefs], ncoefs, d_w,
mu, &d_error[i] );
            }

            cudaMemcpy(w_error, d_error, sizeof(cuDoubleComplex) * Nrows, cudaMemcpyDeviceToHost);
            cudaMemcpy(w, d_w, sizeof(cuDoubleComplex) * ncoefs, cudaMemcpyDeviceToHost);

            cudaEventRecord(stop);
            cudaEventSynchronize(stop);
            float milliseconds = 0;
            cudaEventElapsedTime(&milliseconds, start, stop);
            printf("Execution time elapsed: %f ms\n", milliseconds);

            for(int i=0; i<ncoefs; i++)
            {
                    w_r[i] = w[i].x;
                    w_i[i] = w[i].y;
            }

            for(int i=0; i<Nrows; i++)
            {
                    error_r[i] = w_error[i].x;
                    error_i[i] = w_error[i].y;
            }
    mxGPUDestroyGPUArray(yBB);
}
```

Segunda llamada a CUDA, “cuda_multiply.cu”:

```
#include "mex.h"
#include "gpu/mxGPUArray.h"
#include <cuComplex.h>

__global__ void yBB_Estimation_Post(cuDoubleComplex *yest_post, cuDoubleComplex const *X,
cuDoubleComplex const *W, int ncoefs)
{
        __shared__ cuDoubleComplex lyest_post[256];
        unsigned int tid = threadIdx.x;
        unsigned int stid = blockIdx.x*ncoefs + tid;

        lyest_post[tid].x = 0; lyest_post[tid].y = 0;

        if(tid<ncoefs)
                lyest_post[tid] = cuCmul( X[stid] , W[tid]);    // yBB_est_post=X*w; in MATLAB
        __syncthreads();

        for (unsigned int s=blockDim.x/2; s>0; s>>=1)                   // Reduction
        {
                if (tid < s) {
                        lyest_post[tid] = cuCadd(lyest_post[tid],  lyest_post[tid+s]);
                }
                __syncthreads();
   }
        if (tid == 0)
                yest_post[blockIdx.x] = lyest_post[0];
}

/* Host code */
void mexFunction(int nlhs, mxArray *plhs[],
             int nrhs, mxArray const *prhs[])
{
   /* Declare all variables.*/
   mxGPUArray const *g_X, *g_w;
   cuDoubleComplex const *d_g_X, *d_g_w;
   cuDoubleComplex *d_yBB_est_post;
   cuDoubleComplex *yBB_est_post;
   int Nrows, ncoefs;
   char const * const errId = "parallel:gpu:mexGPUExample:InvalidInput";
   char const * const errMsg = "Invalid input to MEX file.";

   mxInitGPU(); /* Initialize the MathWorks GPU API. */

   /* Throw an error if the input is not a GPU array. */
   if ((nrhs!=2) || !(mxIsGPUArray(prhs[0])))
   {
      mexErrMsgIdAndTxt(errId, errMsg);
   }

   g_X = mxGPUCreateFromMxArray(prhs[0]);
   g_w = mxGPUCreateFromMxArray(prhs[1]);

   /* Verify that g_X really is a double array before extracting the pointer. */
   if (mxGPUGetClassID(g_X) != mxDOUBLE_CLASS || mxGPUGetClassID(g_w) != mxDOUBLE_CLASS)
   {
      mexErrMsgIdAndTxt(errId, errMsg);
   }

   /* Now that we have verified the data type, extract a pointer to the input on the device. */
    d_g_X = (cuDoubleComplex const *)(mxGPUGetDataReadOnly(g_X));
    d_g_w = (cuDoubleComplex const *)(mxGPUGetDataReadOnly(g_w));

    ncoefs = (int)(mxGPUGetNumberOfElements(g_w));
    Nrows = (int)(mxGPUGetNumberOfElements(g_X))/ncoefs;

   /* Create a GPUArray to hold the result and get its underlying pointer. */
   yBB_est_post = (cuDoubleComplex*) malloc(sizeof(cuDoubleComplex) * Nrows);
   cudaMalloc((void**)&d_yBB_est_post, sizeof(cuDoubleComplex) * Nrows);
```

```
    double *yBB_est_post_r, *yBB_est_post_i;

    plhs[0]=mxCreateDoubleMatrix(Nrows, 1, mxCOMPLEX);
    yBB_est_post_r = mxGetPr(plhs[0]);
    yBB_est_post_i = mxGetPi(plhs[0]);

            cudaEvent_t start, stop;
            cudaEventCreate(&start);
            cudaEventCreate(&stop);
            cudaEventRecord(start);

            yBB_Estimation_Post<<<Nrows,256>>>(d_yBB_est_post, d_g_X , d_g_w, ncoefs);

            cudaMemcpy(yBB_est_post, d_yBB_est_post, sizeof(cuDoubleComplex) * Nrows,
cudaMemcpyDeviceToHost);

            cudaEventRecord(stop);
            cudaEventSynchronize(stop);
            float milliseconds = 0;
            cudaEventElapsedTime(&milliseconds, start, stop);
            printf("Execution time elapsed: %f ms\n", milliseconds);

            for(int i=0; i<Nrows; i++)
            {
                    yBB_est_post_r[i] = yBB_est_post[i].x;
                    yBB_est_post_i[i] = yBB_est_post[i].y;
            }

    mxGPUDestroyGPUArray(g_X);
    mxGPUDestroyGPUArray(g_w);
}
```